A COLLABORATIVE VISUALIZATION FRAMEWORK USING JINI™

TECHNOLOGY

THESIS

Chad M. Harris, Captain, USAF

AFIT/GCS/ENG/02M-04

DEPARTMENT OF THE AIR FORCE

AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

Report Documentation Page

Report Date	Report Type	Dates Covered (from... to)
5 Mar 02	Final	Mar 01 - Mar 02

Title and Subtitle	Contract Number
A Collaborative Visualization Framework Using JINI Technology	
	Grant Number
	Program Element Number

Author(s)	Project Number
Capt Chad M. Harris, USAF	
	Task Number
	Work Unit Number

Performing Organization Name(s) and Address(es)	Performing Organization Report Number
Air Force Institute of Technology Graduate School of Engineering and Management (AFIT/EN) 2950 P Street, Bldg 640 WPAFB OH 45433-7765	AFIT/GCS/ENG/02M-04

Sponsoring/Monitoring Agency Name(s) and Address(es)	Sponsor/Monitor's Acronym(s)
AFRL/IFTC Atttn: Dr. Douglas Hozhauer 26 Electronics Parkway Rome, NY 13441-4514	
	Sponsor/Monitor's Report Number(s)

Distribution/Availability Statement
Approved for public release, distribution unlimited

Supplementary Notes
The original document contains color images.

Abstract
It is difficult to achieve mutual understanding of complex information between individuals that are separated geographically. Two well-known techniques commonly used to deal with this difficultly are collaboration and information visualization. This thesis develops a generic flexible framework that supports both collaboration and information visualization. It introduces the Collaborative Visualization Environment (COVE) framework, which simplifies the development of real-time synchronous multi-user applications by decoupling the elements of collaboration from the application. This allows developers to focus on building applications and leave the difficulties of collaboration (i.e. concurrency controls, user awareness, session management, etc.) to the framework. The framework uses an object sharing approach to share information and views between participants in a collaborative session. This approach takes advantage of several Java technologies (i.e. JavaBeans, Jini, and JavaSpaces). JavaBeans establish a well-known standard for applications to operate within the framework. Jini services provide framework stability and enable code sharing across the network. Objects are shared between remote clients through the JavaSpaces service.

Subject Terms	
Collaboration, Information Visualization, Framework, Software, Architectures, Jini, JavaSpaces, JavaBeans.	
Report Classification unclassified	**Classification of this page** unclassified
Classification of Abstract unclassified	**Limitation of Abstract** UU
Number of Pages 130	

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the United States Air Force, Department of Defense or the U.S. Government.

A Collaborative Visualization Framework Using Jini™

Technology

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

In Partial Fulfillment of the Requirements for the

Degree of Master of Science (Computer Science)

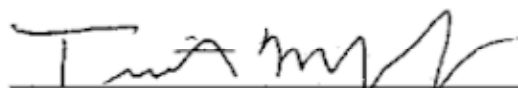Chad M. Harris, B. S.

Captain, USAF

March 2002

A COLLABORATIVE VISUALIZATION FRAMEWORK

USING JINI™ TECHNOLOGY

Chad M. Harris, B. S.
Captain, USAF

Approved:
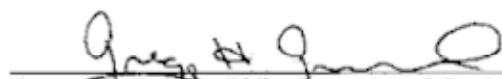
_____   6 MAR 02
Timothy M. Jacobs, PhD, Lt Col, USAF, Committee Chairman   Date
Department of Electrical and Computer Engineering

_____   6 MAR 02
Karl S. Mathias, PhD, Maj, USAF, Committee Member   Date
Department of Electrical and Computer Engineering

_____   6 MAR 02
Gregg H. Gunsch, PhD, Committee Member   Date
Department of Electrical and Computer Engineering

Acknowledgments

Table of Contents

List of Figures

List of Tables

Page

Abstract

It is difficult to achieve mutual understanding of complex information between individuals that are separated geographically. Two well-known techniques commonly used to deal with this difficultly are collaboration and information visualization. This thesis develops a generic flexible framework that supports both collaboration and information visualization. It introduces the Collaborative Visualization Environment (COVE) framework, which simplifies the development of real-time synchronous multi-user applications by decoupling the elements of collaboration from the application. This allows developers to focus on building applications and leave the difficulties of collaboration (i.e. concurrency controls, user awareness, session management, etc.) to the framework.

The framework uses an object sharing approach to share information and views between participants in a collaborative session. This approach takes advantage of several Java technologies (i.e. JavaBeans™, Jini™, and JavaSpaces™). JavaBeans™ establish a well-known standard for applications to operate within the framework. Jini™ services provide framework stability and enable code sharing across the network. Objects are shared between remote clients through the JavaSpaces™ service.

A COLLABORATIVE VISUALIZATION FRAMEWORK

USING JINI™ TECHNOLOGY

I. Introduction

It is very difficult to achieve mutual understanding of complex information between individuals that are separated geographically. Common gestures that are regularly used to communicate one's feelings and emphasize important elements are not entirely conveyed between individuals that are not in a face-to-face meeting. Thus, when separated geographically, other techniques must be used to help individuals communicate and convey ideas. Joint understanding of complex data can be improved through the use of information visualization techniques and interaction with the information through navigation. In this research the concept of visualizing information through a collaborative environment is known as collaborative visualization.

Information visualization techniques present information visually and provide a set of tools to interact with the data, thus allowing for a greater understanding of the information. Collaboration techniques are then used to share and interact with the information and other people to further the mutual understanding of the individuals involved. This capability to enhance mutual understanding is relevant in many areas of industry, education and the military.

One important application of collaborative visualization is in the realm of command and control (C2). Command and control systems provide commanders with critical information to aid them in making decisions. Hidden or obscured information may lead to incorrect decisions resulting in unnecessary damage, injury or death. Collaborative visualization enables experts to uncover the hidden information and provide it to commanders so decisions are made that minimize risk and maximize desired outcomes.

1

Teams or divisions of people are found in almost every facet of life, and certainly when these people are separated geographically, communication and collaboration barriers can be created. This holds true in a combat situation where many different systems must work together to provide leaders with information to guide their decisions. The creation of nonintegrated repositories of data or hardware components, each designed for a specific purpose that is not reconciled with each another is the concept referred to as "stovepipe" systems. This creates serious problems for the logistics communities due to the uniqueness of systems within the military. Providing a mechanism for teams of engineers to collaborate and share ideas and provide mutual understanding could greatly increase the military's ability to develop and sustain new weapon and support systems.

In the education arena, distance learning and satellite education and training are becoming increasingly more popular. The goal of such programs is to provide educational and training services to people across the globe and to bring people together to collaborate and exchange ideas and work together to edify each other. Collaborative visualization has similar goals in that we want to provide a mechanism to increase understanding through visualization and interactive communication. The military can use these techniques to train people all over the world, saving the government money in travel and accommodation costs.

The importance of collaborative visualization can be seen in all walks of society, and especially in the Department of Defense (DoD). The focus of this thesis is the application of collaborative visualization techniques to develop a generic framework to support the command and control arena.

1.1 Background

The current capabilities of combat weapon systems are limited in many different ways. They involve labor-intensive collection and coordination processes to disseminate relevant information to commanders and people within a battlespace. Current processes and systems gather great amounts of data, but the ability to make use of that data is limited, thus creating an environment for information starvation. Information starvation is the inability to make effective use of the current information already gathered,

thus commanders do not have the necessary information needed to make advantageous decisions. In addition, the immense amount of data causes information overload, due to the fact that there is too much data to process. Currently there is little interoperability in joint and coalition force systems, which creates difficult circumstances to integrate information and provide a synergy for operations. Figure 1 typifies how our current combat systems have worked in the past. Interoperability exists between some systems, but little cross-flow of information between systems in the horizontal direction is present. The lack of cross-flow creates systems that are nonintegrated and designed for specific purposes. The lack of interoperability creates these "stovepipes" and they hamper the effectiveness of forces to fight and win battles. Disjoint workflows and wasted man-hours result from this lack of synergy. The Department of Defense and other government and private agencies collect information every day. This information is being underutilized, often duplicated, and not fused together to create a seamless current overall view of the battlespace. This hinders the ability of a commander to make the right decision in the right time.
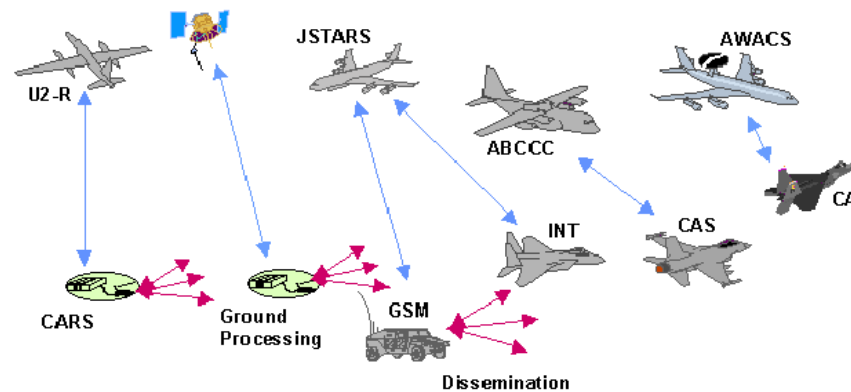
Figure 1. Current Combat Information Reality [57]

The United States Air Force Scientific Advisory Board presented the concept for the Joint Battlespace InfoSphere (JBI) in the report *Information Management to Support the Warrior* [57]. This concept was envisioned to help overcome these challenges that face our combat systems today. The JBI "is a combat information management system that provides individual users with the specific information needed to accomplish their functional responsibilities during a crisis or conflict" [48]. Its main challenge is "providing the right information at the right time, disseminated and displayed in the right way, so

commanders and crew chiefs can do the right things in the right time in the right way" and do it faster than the enemy [48]. The vision of JBI is to integrate or fuse current combat systems in a seamless environment known as the "Battlespace InfoSphere", as depicted in Figure 2.
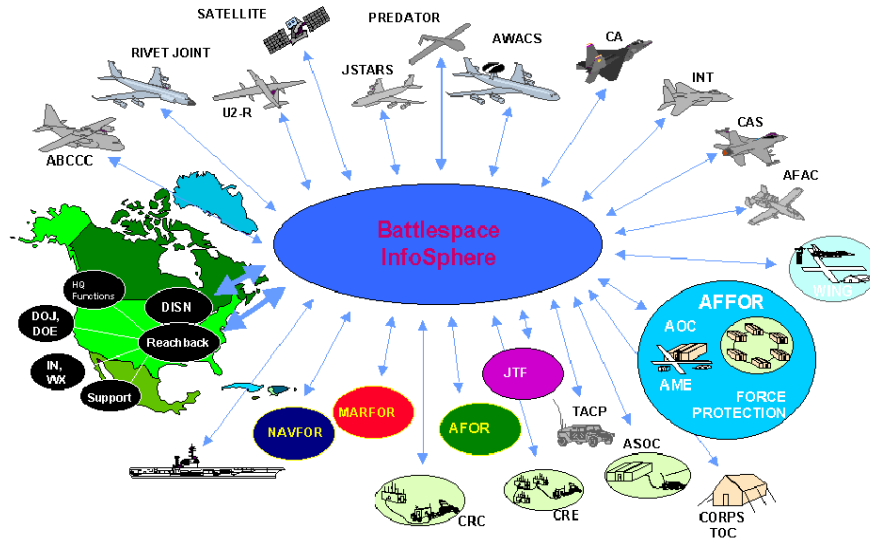
Figure 2. Joint Battlespace InfoSphere of Tomorrow [57]

The Battlespace InfoSphere can integrate information systems of today and tomorrow and should provide the synergy needed for the future. Combat operations in the future are likely to depend on integration of information systems, due to the need to share information. This is evident today due to the events of September 11, 2001 in New York City, New York. Current military operations in Afghanistan and throughout the world depend on shared information sources from around the world. Further JBI discussion follows in Chapter II.

1.2 Problem Statement

The focus of this research is to develop a flexible collaborative visualization framework for integrating tools (implemented as JavaBeans™) into an environment that enables collaboration between geographically separated users. Through a review of related work, the following characteristics have been identified as important for such a framework: automation, shared state, consistency, scalability,

communication, robustness, dynamic loading, coupling, language support, flexibility, coordination of action, monitoring, and protection. These characteristics are discussed in detail in Section 3.1.1.

A comparative analysis between the developed framework and others is performed to evaluate the overall effectiveness of the framework. The criteria for the evaluation is a measure of how well each framework fulfills the characteristics listed above.

1.2.1 Goals

The overall goal of this research is to create a framework that supports both collaboration and information visualization techniques. To measure the success of the framework, the following sub-goals are established:

- Shared interaction – the collaboration between geographically separated users interacting with data and visual representations to accomplish tasks. The characteristics used to measure this goal are: shared state, data consistency, communication, coordination of action, monitoring, and protection.

- Visual sharing – remote users collaborate at a higher level of abstraction than the data through the sharing of visual objects. The criteria established to measure the effectiveness of this goal are visual consistency and visual collaboration.

- Remote code access –users access remote data and applications without the need for previous installation. Dynamic loading is used to measure the effectiveness of the framework in meeting this goal.

- Easy tool integration – flexibility to easily integrate any Java tool and make it collaborative. To measure this goal the following criteria are used: generality, automation, and coupling.

- Facilitate software development – the purpose of a framework is to ease the burden of developers in developing software. The following characteristics are defined to measure the

capability of the framework: scalability, robustness, standard language support, and flexibility.

Fayad, Schmidt, and Johnson [18] added that a framework must be modular, extensible, and reusable. Modularity is the division of software into logical components, thus reducing the complexity and effort required to understand and maintain software. Extensibility is the ability for software to be extended to meet the needs of new users. This allows for customization of new applications and services. Reusability enables generic components to be mixed and matched to produce the desired functionality of an application.

With the understanding of what makes up a good framework, the framework developed in this research must include common elements of a collaborative system along with support for information visualization techniques. Some key elements of a collaborative system as defined by Gutwin and Greenburg [22] are: explicit communication, consequential communication, coordination of action, monitoring, assistance, and protection. These concepts are discussed further in Chapter II. These collaborative components provide a good measure to ensure this framework supports the necessary collaborative objective. In addition, the framework should include a set of information visualization tools and techniques to display large sets of data. The techniques used in the framework need to be generic to support many different types of information.

In summary, this research should produce a modular, extensible and reusable collaborative visualization framework. This framework should be able to share information in a common space, and enable communication of individuals effectively and efficiently, in addition to providing effective information visualization tools to display large quantities of data.

1.2.2 Constraints

There are a number of limiting factors imposed upon this development and research effort. Jini™ technology is the primary focus for communication between hosts and provides the mechanism for object sharing. This restriction is due to the desires of the sponsor, Air Force Research Labs, Information

Technology Division (AFRL/IFTC), to use Jini™ technology as one of many middleware technologies as part of the backbone for the Joint Battlespace InfoSphere. Additionally, Jini™ technology provides the desired capabilities needed in the distributed collaborative environment.

One of the desired capabilities of the framework is to enable collaboration at a higher level of abstraction than that of the data. This concept of visual sharing has the potential to require greater amounts of bandwidth due to the need to send entire objects across a network instead of small data changes. Such bandwidth issues are not explicitly considered in the design of the framework.

To develop a framework that supports every type of application and programming construct without any constraints would be an arduous task if not impossible. Thus to limit the set of applications, the JavaBeans™ [52] component architecture is used. This allows for a generic solution to be developed by restricting the possibilities of applications to a well-defined component architecture. Thus, the design of the framework can utilize the JavaBeans™ technology to its advantage.

Voice and video collaboration is not addressed as part of the research, due to the fact that Jini™ does not readily support data streaming. Commercial-off-the-Shelf (COTS) products, like Windows NetMeeting and others, may be appropriate to use for video and voice communications.

Consequential communication is information that is "given off" unintentionally by others as they go about their activities. This information is communicated through the manipulation of artifacts or the characteristic actions of a person in a workspace [22]. Some consequential elements are handled by the use of annotation tools within a whiteboard environment. Finger pointing can be simulated by mouse annotation, thus covering a small element of consequential communication. This research does not address every consequential communication element of a collaborative system. Much research is needed in this area to be able to capture facial gestures, finger pointing, and other forms of body language. Video streams could be utilized to capture this type of information, but are not considered in this research.

1.3 Document Overview

The first chapter describes the overall objectives of this thesis and establishes the importance of information to the JBI concept. Chapter II provides an in-depth discussion of background material related to this thesis, with focus on JBI concepts, collaborative techniques and technologies, visualization techniques and frameworks design. Jini™ technology is discussed extensively in this chapter to provide some background information on why it was chosen as the potential solution for backbone communication. Chapter III discusses the problem approach and design of the framework itself. Chapter IV discusses the implementation of the framework and provides examples of its application. Chapter V summarizes the results of the overall effectiveness of the framework along with its impact on the JBI concept. Chapter VI provides conclusions and recommended areas for future research.

II. Background

This chapter provides background information to help in understanding why certain technologies and approaches are taken in this research. This chapter discusses the domain in which this research is relevant, along with related research.

2.1 Command and Control (C2) Domain

Command and Control (C2) systems are essential for successful completion of military operations. C2 can be defined as "the process of gathering information, assessing situations, identifying objectives, developing alternative courses of action, deciding on a course of action, transmitting orders, and then monitoring execution" [34]. Based on this understanding, C2 systems must have the ability to process large amounts of data in a short period of time and present a "big picture" to the commander. This may require collaboration with many units, agencies and services within a battlefield to collect all the necessary information. Information visualization techniques can be used to process immense quantities of data quickly and display it in a manner to help commanders understand this data. In addition, collaboration techniques aid the convergence of ideas to a "best" decision more quickly because ideas are being shared simultaneously. These techniques enable command and control systems to collaborate information at many levels of command, from the Commander in Chief to the infantry soldier out on the front lines of battle.

C2 systems also rely on many different types of simulations, which require great amounts of data processing. The outputs of these simulations are difficult to understand due to the amount and complexity of the data, so visualization can be performed to make sense of the information. These visualizations may become large in size; thus, the ability to handle transmission of complex information is essential.

2.1.1 OODA Loop

A key aspect in any command and control system is the ability to orient oneself to a situation and then make a well-informed decision to gain an advantage over an enemy. Col John R. Boyd generalized

this process in 1987, in his paper "Patterns of Conflict" and called it the OODA loop model. This model consisted of four basic steps: Observe, Orient, Decide, and Act (OODA). This model defines the cognitive process of how decisions are made and indicates the activities that go on to produce actions from circumstances. Many branches of military service have adopted this model and incorporated it in their command and control doctrine due to its simplicity and completeness. This model essentially describes the functions that C2 systems must provide to commanders to aid in the decision making process.

As described by Grant T. Hammond,

Knowledge of the strategic environment is the first priority. Secondly, one must be able to interact with the environment and those within it appropriately. You must be able to observe and orient yourself in such a way that you can indeed survive and prosper by shaping the environment where possible to your own ends, by adapting to it where you must. Doing so requires a complex set of relationships that involve both isolation and interaction. Knowing when each is appropriate is critical to your success. In OODA Loop fashion, one must continually observe, orient, decide and act in order to achieve and maintain freedom of action and maximize the chances for survival and prosperity. One does so through a combination of rapidity, variety, harmony, and initiative. It is these concepts that are the core of 'Boyd's Way.' Rapidity of action or reaction is required to maintain or regain initiative. Variety is required so one is not predictable, so there is no pattern recognition for a foe to allow him to know of your actions in advance and thus plan to defeat them. Harmony is the fit with the environment and others operating in it. Initiative—taking charge of your own destiny—is required if one is to master circumstances rather than be mastered by them. All of course, would be focused on attaining the specified Objective that is implicit in this discussion. [25]

To understand the OODA loop process, each component is described below.

The first element, (**O**)bservation, is the process of taking in and absorbing one's environment. This view would be entirely empirical if the observer could guarantee the reliability and objectivity of the sensors viewing the environment. The second element, (**O**)rientation, is the most important step in the loop. [Orientation is process by which data is simulated and processed into information and understanding. In other words, it is the sense-making phase of the model.] It is the most easily corruptible of the four steps. Orientation requires the observer to yield to frail human qualities, such as culture, heritage, and, most importantly, previous experience. This is one place in the cycle where there is feedback from previous evolutions. Orientation may be drastically altered based on the experience of success or failure from a proceeding evolution. The third element, (**D**)eciding, is the cognitive process of selecting a course of action among the options that present themselves from the observation and orientation portions. As Boyd [writes], 'In short we engage in a complex process of analysis and synthesis before selecting a course of action … we assess a variety of competing, independent channels of information from a variety of domains to cope with the particular circumstance which confronts us.' The final element, (**A**)ction, is simply doing the course of action selected in the decision portion of the cycle; however, in some instances it is the most difficult to implement. [9]

Figure 3 shows how each phase works together within the model to produce action from the information gathered. In this figure orientation implicitly provides guidance and control to shape observation, decision, and action. Orientation is in turn, shaped by the feedback and other phenomena coming into our sensing or observation window. This is why the orientation phase is the most important and volatile of all phases. Orientation accounts for cultural traditions, genetic heritage, previous experiences, and new information. Each facet accounts for the different element that affects our orientation.
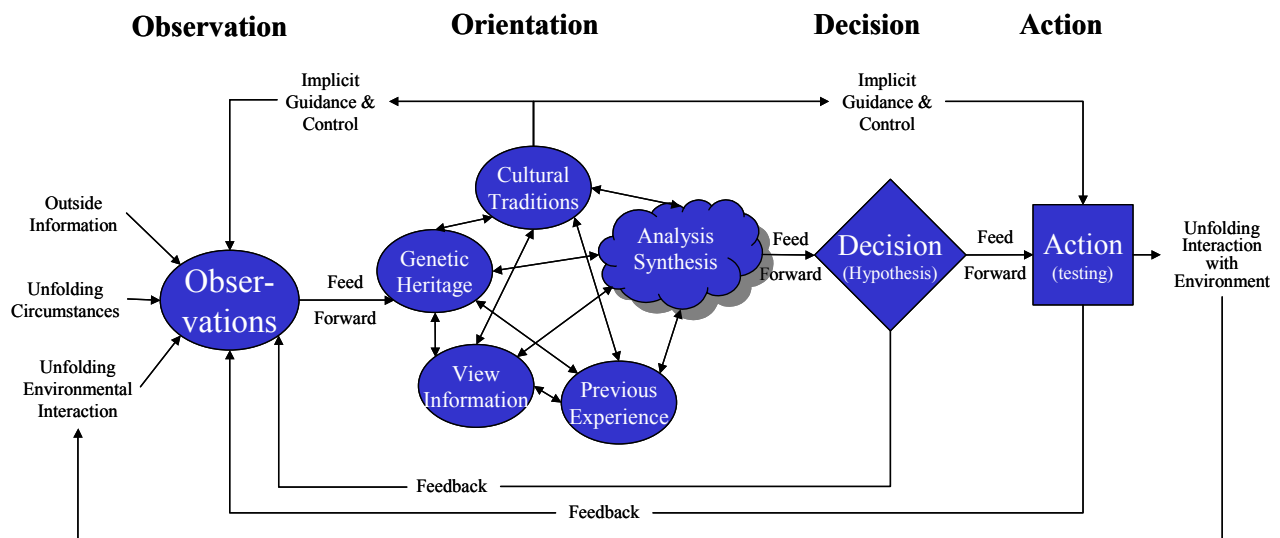
Figure 3. OODA Loop Process [2, 37]

Cultural traditions and genetic heritage provide rooted experience and customs that form a frame of reference from which to orient our minds. Previous experience provides invaluable historical perspective when placing information into context. Hopefully we learn from previous mistakes and history and make decisions based on what worked and what didn't work. New information changes the way we orient ourselves because it brings new light to the situations at hand. By having additional information, commanders can adjust to the circumstances and ensure that the decisions they make minimize casualties and maximize effectiveness.

2.2 Visionary Concepts

To help understand the focus of this research it is imperative to understand the underlying theory and vision documents that guide the Department of Defense in accomplishing its mission.

2.2.1 Joint Vision 2020

Joint Vision 2020 is a visionary document that describes the Chairman of the Joint Chief of Staff's (CJCS) vision for future military joint forces. Our military force has a primary purpose to fight and win the nation's wars. This requires a vision of "Dedicated individuals and innovative organizations transforming the joint force for the 21st Century to achieve full spectrum dominance – persuasive in peace, decisive in war, and preeminent in any form of conflict" [11]. This enables the United States to protect our interests around the world and our nation's freedoms.

To achieve full spectrum dominance, the military must continue to invest in and develop new military capabilities. The continued development and proliferation of information technologies will change the conduct of military operations. Our ability to achieve full spectrum dominance makes information superiority a key enabler to change the joint command and control system for the future.

Information was the key to victory throughout history. However, information superiority is only useful when it is effectively translated into superior knowledge and decisions. "The joint force must be able to take advantage of superior information converted to superior knowledge to achieve 'decision superiority' – better decisions arrived at and implemented faster than an opponent can react" [11].

The concept of the global information grid provides a network-centric environment that provides global interconnectivity which will enhance combat power. This information-sharing environment for the future will require not only technological advances, but also changes in policy, organization structure and doctrines to achieve the goal. Thus, to help reach the goals set forth in Joint Vision 2020, a mechanism for information sharing is needed. This research focuses on creating a framework that enables an information-sharing environment.

2.2.2 Air Force Vision 2020

In response to the Joint Vision 2020, F. Whitten Peters, the Secretary of the Air Force, and General Michael E. Ryan, United States Air Force Chief of Staff, produced "America's Air Force Vision 2020" [10]. This document provides a vision to how the United States Air Force will meet the demands of the future.

> In a world that is globally connected, national security and international stability are vital foundations for America's prosperity. Assuring security and stability require global vigilance, reach and power – global vigilance to anticipate and deter threats, strategic reach to curb crises and overwhelming power to prevail in conflicts and win America's wars. [10]

With these defined objectives the United States Air Force will realize full spectrum dominance envisioned by Joint Vision 2020.

To meet these objectives requires dedicated men and women who lead effectively at all levels, resources necessary to gather information and provide command and control support, and superior weapon platforms necessary to accomplish missions. The goal of this research is to develop support tools that will aid leaders at all levels to make good decision and share information.

"Information superiority will be a vital enabler of that capability" [10]. This goal of information superiority requires technology to gather, fuse and integrate new and existing systems to enable unprecedented access to information. "We'll rely increasingly on distributed (or reach back) operation to effectively sustain our forces, providing time-definite delivery of needed capabilities. Fast, flexible, responsive, reliable support will be the foundation of all Air Force operations" [10]. The reliance of distributed systems will require all types of collaboration in the future. This research is geared to help in providing a technological solution to visual collaboration for command and control systems and other relevant military domains.

2.2.3 Joint Battlespace InfoSphere (JBI)

Information superiority is a recurring theme in the Joint Vision 2020 and the Air Force Vision 2020. As was stated in the introduction, the challenge is "to provide the right information at the right time and disseminate or display it in the right way so commanders will act in the right way at the right time to defeat their adversary" [57]. Information superiority is essential if this objective is to be met. Therefore, the United States Air Force Scientific Advisory Board developed the concept of a Joint Battlespace Infosphere (JBI) as the vision for future information systems. "The JBI is a conceptual combat information management system that provides individual users with the specific information required for their functional responsibilities during crisis or conflict" [35]. "The Battlespace Infosphere (BI) must provide integrated mission understanding, shared awareness, shared planning, shared execution, shared visualization, shared support, and [a] shared future view" [35].

2.2.3.1 JBI Basic Concept

The JBI objective is to provide a mechanism to interpret information and make decisions faster than the enemy, thereby ensuring information superiority. Current legacy information systems provide much information to today's combatants, but they are disjoint and poorly organized. A network-centric approach to information systems must be achieved to support joint and coalition operations.

> Network-centric warfare is a first step in the direction of forming a common view of the battlespace by ensuring ubiquitous connectivity. Network-centric systems gain their operational advantage by integrating existing planning and warfighting systems via a communications network. The BI extends the concept of the network-centric system. It remains essential that existing and evolving function-specific systems be interconnected and able to intercommunicate. But in the BI, capabilities for intelligent data transformation, information exchange, knowledge sharing, and processing provide the operational advantage. [35]
>
> The BI provides a highly tailored repository of, or access to, information that is designed to support a particular geographic area or mission. The intent of the BI is to have a 'single place,' a 'virtual system of information systems,' that serves as a clearinghouse and a workspace for anyone contributing to the accomplishment of the operation—for example, weather, intelligence, logistics, or personnel. The use of the BI seamlessly integrates multiple sources of data, enables automated manipulation of data, provides faster response times, and produces tailored information to support warfighter decision making throughout all functional staff activities. [35]

There are three main broad categories that allow for interaction with the BI: input, manipulation and interaction. Information must be placed in the InfoSphere, followed by its manipulation to create knowledge, and finishing with people gaining access to the knowledge. The basic functions of each category are depicted in Figure 4.
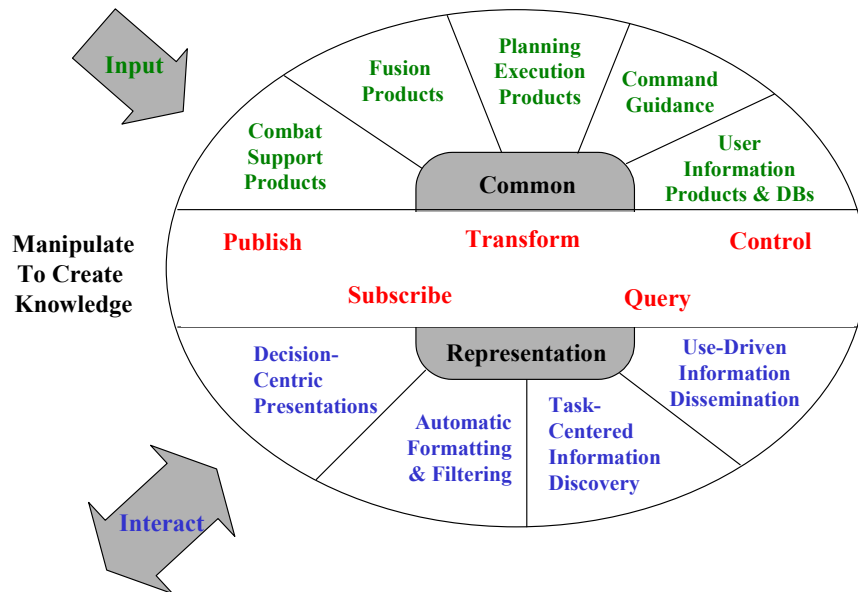
Figure 4. Components of Battlespace InfoSphere [57]

Each element in the BI is an object that encapsulates some information about the battlespace. Objects are input into the BI and are made available for manipulation within the BI. Objects may be manipulated in five ways: publish, subscribe, transform, query and control. Publishing objects makes them available to communities for distributed use. Users wishing to use these object then subscribe to them and are thus provided with the information these objects represent. A transformation operation is needed to convert objects from one format to another. The query capability allows the large repository of information stored in the BI to be searched. Control provides a mechanism to administer and organize the objects. These actions provide the mechanism for users of the BI to manipulate the information and gain understanding of the battlespace. This figure shows how input is received via many different forms (i.e. Combat Support, Fusion, Planning Execution products, etc.) and compiled into a common representation.

The five manipulation mechanisms enable information to be composed into a common format, as well as enabling users to interact with the information. Users interact with the common representation through an application that supports different strategies (e.g., Decision-Centric Presentations, Task-Centered Information Discovery, etc.), thus enabling the manipulation of information to create knowledge.

The outcome of this research provides a small piece in the information sharing of the JBI concept. This idea of providing mechanisms for users to interact with information is key to the success of the JBI. A framework that supports collaborative visualization will add another capability desired in the JBI. With this framework, task-centered information and data-centric presentation can be aided.

2.2.3.2 Enabling Technologies

The vision of the JBI will require many enabling technologies to make this possible. As one small part of the JBI effort, Air Force Research Labs, Information Directorate, Rome Labs is looking into the feasibility of integrating military data into a Jini™ based network. Jini™ technology is a sophisticated platform to build network-aware applications. This technology makes it possible for users to access resources located anywhere on the network. In addition, network resources and devices can join and leave the network without any human intervention or manual configuration. The focus on Jini™ is due to its robustness as a network protocol. The focus of this research will help in this effort by using Jini™ as a backbone for a collaborative visualization framework.

2.3 Application Frameworks

An application framework is a "reusable, semi-complete application that can be specialized to produce custom applications" [18]. Application frameworks describe both the components and how each component interacts. Another definition, described by Gamma, is "a set of cooperating classes that makes up a reusable design for a specific class of software. A framework provides architectural guidance by partitioning the design into abstract classes and defining their responsibilities and collaborations" [18].

2.3.1 Object-Oriented Techniques

Object-oriented (OO) techniques make design and implementation of application frameworks much easier. They allow for the definition of abstract classes or components that provide building blocks for an application framework. OO application frameworks take advantage of data abstraction, polymorphism, and inheritance. Using data abstraction, the framework can present an interface for a class and hide all the underlying details. Polymorphism lets the developer mix and match components, lets the object change its collaboration at runtime, and makes it possible to create generic objects that can work in many different applications. Object hierarchies can be developed that give developers the ability to customize components from a framework in their specific applications.

A common observation made about framework design is that it takes iterations to fully reach the maturity of its intended purpose. Additional information is inevitably discovered later in the development process, which leads to iterations. Frameworks usually implement the explicit parts of the design that are likely to change, forcing iterations to occur. Frameworks are generally large sets of generic components that are collected and developed over time from sets of desired functions to suit the needs of a problem. It is impractical to analyze and design every conceivable scenario prior to implementation. Current trends are to build the framework iteratively by adding additional capabilities and components later on in its life cycle [18].

2.4 Collaboration

For purposes of this research, collaboration is defined as "two or more geographically dispersed individuals working together to share and exchange data, information, knowledge, and actions" [36]. The product of collaboration is loosely defined and may include a decision, a document or any outcome from the interaction between the collaborants. "A collaborant is a resource that participates in a collaboration and that can reason about and take action on the state of evolution of the products and processes" [44]. The collaboration environment consists of the necessary resources to allow for communication between individuals to take place.

17

This research focuses on the development of a collaboration framework, which is "the necessary and sufficient set of standards-based computing and communication infrastructure, and collaboration support services used to develop and execute instances of a collaborative environment" [44] that supports visualization.

2.4.1 Types of Collaborations

2.4.1.1 Asynchronous Collaboration

Jason Wood presented the idea of asynchronous collaboration visualization in which people interact at different times, in different places…such as when sending email messages or leaving messages on a bulletin board [59]. This type of collaboration may be more indicative of development environments and business practices, but is not necessarily the best approach for C2 systems.

2.4.1.2 Synchronous Collaboration

Most C2 systems are manned 24 hours a day, 7 days a week, and have the need for synchronous collaboration with immediate responses. The immediate response and direct interaction between collaborants is the big difference that distinguishes asynchronous from synchronous collaboration systems.

2.4.2 Levels of Collaboration

Asynchronous and synchronous collaboration can be achieved at several different levels. This may range anywhere from the data level to the view level. Data collaboration uses raw data to communicate information from one place to another, such as, sharing a file over a network. View collaboration is sharing a visual representation to communicate ideas and information between collaborants, such as, sharing an external window with another computer. Figure 5 shows the different levels of collaboration that can occur. An example of data abstraction is the sharing of data objects or structures. Likewise, the visual abstraction is the sharing of visual objects that compose the view.

18

Levels of Collaboration and Abstraction
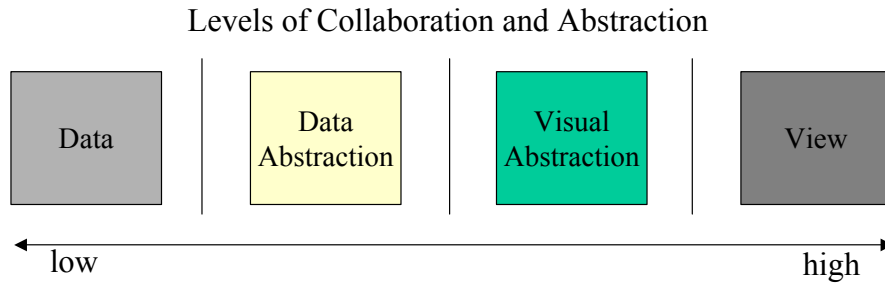
Figure 5. Levels of Collaboration

Most collaboration systems currently use the data abstraction level to enable collaboration between users. The primary use of data abstraction is to create message objects that hold some information about the collaborative components. These messages are passed back and forth amongst the collaborants and form the basis for collaboration.

2.4.3 Elements of Collaboration

Many elements are required for effective collaboration. Gutwin and Greenberg [21, 22] identified several mechanisms of collaboration that must be present when working in a shared workspace. They are: explicit communication, consequential communication, coordination of action, monitoring, and protection.

- Explicit Communication: Verbal and written communications are the cornerstone for sharing information in a collaborative environment. Within a visual workspace, the workspace and the artifacts themselves are critical in enabling explicit communication to take place.

- Consequential Communication: In addition to explicit communication, great amounts of information are transmitted and picked up unintentionally as people go about their activities. The manipulation of objects from within a collaborative session produces two kinds of information: consequential communication and feedthrough. "Consequential communication is the visible or audible signs of interaction with a workspace" [23]. For example, watching someone work provides clues about their actions. "Feedthrough is the observable effects of someone's actions on

the workspace's artifacts" [23]. For example, seeing an object move indicates that someone is moving it.

- Coordination of Action: To avoid conflicts within a shared environment, participants must take turns. In addition, some tasks may require a sequence of events to take place in a certain order. Thus, it is imperative that collaborants can organize their actions within the shared workspace so conflicts do not arise. This implies that some type of control mechanism must be in place to give control of the shared workspace to users.

- Monitoring: Almost all other mechanisms for collaboration rely on the ability to monitor and collect information about others in the workspace. Workspace awareness information, such as who is in the workspace, where they are working, and what they are doing, is primarily the information needed to enable collaborants to make progress on tasks and effectively work together.

- Protection: One obvious potential problem with a shared workspace is the inadvertent altering of work by another in the workspace. Thus, people must be courteous of others' work and not inadvertently destroy someone else's work. In addition, mechanisms for locking different elements or regions of the shared workspace with passwords may be necessary to provide adequate protection to a user's work.

2.5 Information Visualization

Information visualization is a proven concept used to help in understanding information. This section discusses the process of understanding and relevant visualization techniques that are useful in aiding understanding. The framework to be developed should support information visualization.

2.5.1 Knowledge Crystallization

Card, MacKinley and Schneiderman [4] present the idea of knowledge crystallization, which is the process by which we gain understanding of information. It is similar to Boyd's OODA loop and it assumes five simple steps. The steps are:

- Information foraging

- Search for schema

- Instantiate schema

- Problem-solve

- Author, decide or act

During any one of these steps, certain tasks are performed to aid in the cognition of that particular step. To help understand some tasks that may be performed in this process, some techniques employed in information visualization are discussed.

The first step in the process is to forage for information. The primary reason is to gather enough data to be able to make some sense of the data. Visualization techniques (e.g., navigation techniques) are proven in aiding the forage for data. These techniques allow a user to interact with information to gather bits and pieces of facts that will help solve the task or problem at hand.

Once the data has been gathered, making sense of the information follows. This involves a process of mapping the information to a meaningful representation (schema) so it makes sense in human minds. If data does not map well to a particular schema, the search for a more suitable schema should proceed, thus producing a schema that can abstract the information so it can be managed and processed. Some techniques that are used to instantiate schemas are: reorder, cluster, classify, average, promote, and pattern detection.

With the data mapped to a local representation, the process of problem solving can commence. This is the process of instantiating solutions and comparing the effectiveness of the potential results. Visualization techniques that are used are: read fact, read comparison, read pattern, manipulate, create and

delete. As irrelevant information is deleted, reorganized and compared, solutions to problems are more easily apparent.

Once the problem-solving phase reduces the number of solutions, the last step is to take action. The action does not always have to require a decision or action, but may include a report, a briefing or another type of document that composes information.

This knowledge crystallization is used in all domains, and information visualization techniques have proven very useful in helping the human cognitive process come to resolution more quickly. Patterns are more easily seen and information is presented in a way that allows the human to make sense of the info and see how it fits in context of a problem. Several techniques are discussed to demonstrate the usefulness of these techniques in knowledge crystallization.

2.5.2 Maps

Almost everyone has used a map to navigate from one place to another. The primary advantage of maps is that they provide a high density of information in a small space. Distances can be calculated, and size of cities and political boundaries can be assessed and considered when making decisions. Many times within the C2 community, political boundaries must be understood and taken into consideration when deciding on a manner to attack an enemy or diffuse a situation. Maps can provide this type of information at a glance. Imagine trying to understand a nation's political boundaries, just by looking at raw latitude and longitudinal data. The visualization provides an image of the political boundaries around nations and states, so the user only has to glance at a picture to gain an understanding.

2.5.3 Symbols

Symbols or Glyphs are another important visualization technique that is used within the C2 realm. "Glyphs are graphical objects or symbols that represent data through visual parameters that are either spatial, retinal, or temporal" [8]. Symbols abstract information into simple meaningful entities. Some may argue what "meaningful" truly is, but within the C2 community, certain symbols have evolved to represent

information. These glyphs may not be the most intuitive or best-designed symbols, but they do provide an abstraction (e.g., infantry, artillery) that can be composed to aid understanding.

Within the C2 community, common symbols are composed of three components: an icon, a frame and a fill. Icons are glyphs that represent units, location or equipment. To illustrate this, Figure 6 provides an example of two different units. The following symbols and charts are taken from the United State Army Field Manual 101-5-1 [12].

This figure shows two icons, one being a flag annotated with TOC, and the other being a square with a dot in the middle. In our culture, the flag has an understood meaning of some type of headquarters or an important unit. In this case, the flag is annotated with the use of a label (TOC) to indicate that it is a Tactical Operations Center. The other symbol is an artillery division. The X above the artillery division indicates its size. The X represents a Brigade; thus, this unit may have many subordinate units within it. It is arguable how intuitive these symbol are, but these symbols have been used for years and have become part of the C2 community and are understood within that domain. The use of labels is discussed in the next section, but it is important to point out that if there were no label associated with the icon, it would be very difficult to distinguish one icon from another. These icons provide an abstraction to be used on a map that hides the details from the user. These details could be explored by using a drill-down technique that is discussed later, to find more information about these units.

Tactical Operation Center (TOC)   Artillery Division
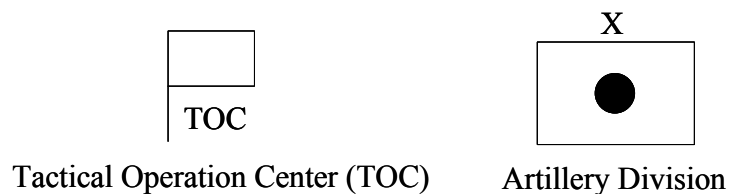
Figure 6. C2 Symbols Example

The next component of a common C2 glyph is the frame, which consists of primitive shapes and symbols that present affiliates. Table 1 shows a representative group of frames that are used within the C2 community. These frames are "role indicators" that show the warfighting function the unit performs either on the ground, air, or at sea.

	Friendly Ground Units	Friendly Sea/Air	Unknown Sea/Air	Neutral	Enemy Units
Surface					
Subsurface					
In-flight					

Table 1. Unit, Installation, and Site Symbol Frames [12]

The last component of the glyph is the fill, which refers to the area within a frame. The fill is usually filled in with color, but does not have to be used. Color is discussed in a later section.

2.5.4 Labels

As was mentioned above, labels are a critical element to be used with symbols. In Figure 6 it is impossible to tell what artillery division this is. To avoid this ambiguity, labels are used to indicate unit designation. These labels provide additional information about the unit to help paint the bigger picture.

Labels are also needed to point out landmarks and other relevant items on a map. Imagine trying to look at a map with none of the cities or highways labeled. It would be difficult to find information and understand it, especially for one unfamiliar with the landscape. To resolve this problem, labels are used to provide visual cues about location. This allows individuals to understand context more quickly and then orient themselves to the situation.

2.5.5 Color

Color is a very useful technique when trying to distinguish and group objects. Making common objects look the same can have important meaning and provide additional abstract information. Examine

the use of color to show highway types. On a common street map, principal highways are usually red, while interstates are blue. This is very helpful when these roads intersect, because it enables one to follow the roads without getting lost in the intersection.

As mentioned above, the fill of a symbol is usually done with color. Color indicates affiliation. The C2 community has a standard color representation, shown in Table 2. By using these colors, additional, but possibly redundant, information is presented. Overall, the use of color is very helpful in distinguishing one object from another. On a map, coloring the land green and the sea blue makes it easier to see the distinction of information on the display. Plus, the use of color makes a display much nicer to look at and easier on the eyes if used appropriately.

Affiliation	Color	
	Hand-Drawn	Computer-Generated
Friend, Assumed Friend	Blue	Cyan
Unknown, pending	Yellow	Yellow
Neutral	Green	Green
Enemy, Suspect, Joker, Faker	Red	Red

Table 2. Color Representation [12]

2.5.6 Interaction

Another key visualization technique is being able to interact with data. It is nice to see the information in a big picture, but if additional data is needed, there must be ways to obtain the data easily. Some techniques that allow users to interact with data, such as drill-down, filtering and zoom, are discussed next.

2.5.6.1 Overview and Detail

Overview and detail allows the user to interact directly with the data to request more detail, while maintaining a high level view. By so doing, more variables can be shown about the information, which aids understanding.

An important aspect of this technique is the ability to drill down or obtain details-on-demand. This capability allows users to obtain additional information by displaying more details in different views. In the C2 community, this enables getting information about units and equipment very easily. Especially on the logistics side of C2, the commander must understand what each unit is supplied to do. If a mission were going to take a certain amount of ammunitions, it would be wise to see if that amount of resources is available. Additionally, the commander may want to know who is the most suited for the task, based on their current supply and status. Thus, by providing the commander with additional details about a unit, the best decision can be made.

2.5.6.2 Filtering

Many data points may be overwhelming to any commander. Thus, the ability to mask unimportant data and focus on relevant information is imperative while trying to forage for data. Thus, by eliminating extraneous data, more time and attention can be given to the smaller subset of important data. With fewer things on one's mind, the cognitive process is achieved more effectively.

2.5.6.3 Zoom

The zoom capability enables users to focus in on specific details of a view by enlarging a section of the view. This visualization technique allows users the ability to enlarge the view to extract additional information at a smaller level of granularity. This capability allows the user to zoom in for more detail and zoom out to access the big picture. This technique helps achieve understanding of information.

2.6 Distributed Systems

Collaborative systems enable multiple clients to share common views or data to accomplish a task. This, in essence, is a distributed system of clients working together to solve a problem or task. By making this parallel, it becomes apparent that there are many complexities that come into play to facilitate collaboration.

Peter Deutsch recognized the challenges of distributed systems over that of classical standalone systems and wrote "The Seven Fallacies of Distributed Computing" [13]. He added, "Essentially everyone, when they first build a distributed application, makes the following seven assumptions."

- The network is reliable

- Latency is zero

- Bandwidth is infinite

- The network is secure

- Topology doesn't change

- There is one administrator

- Transport cost is zero

Some additional assumptions to consider are:

- The network is stable

- Resources are infinite

- The network is homogeneous

"All prove to be false in the long run and all cause *big* trouble and *painful* learning experiences" [13]. Distributed systems must implement mechanisms to handle these issues. An effective and efficient collaborative framework is no different and must handle these types of concerns. A few distributed concepts such as performance and latency, failures, concurrency and consistency are briefly discussed to provide some background information.

2.6.1 Performance and Latency

Clearly accessing resources – data or files – over a network is slower than accessing them locally. In some cases the difference may be orders of magnitude, which in turn affects the overall performance of an application. This difference in performance, in a distributed system, becomes an issue due to the inability to distinguish a failed component from a slow component. Thus, the problem of handling failures

improperly is perpetuated. The performance of a network may vary greatly within a short span of time. This is evident if one has ever surfed the web. Thus, a distributed system must be able to handle a variety of network conditions. Developers of distributed systems often overlook and even ignore performance and latency concerns because remote accesses are made to look like local ones. This results in applications that have communication patterns that are unacceptably slow, or not very robust in a distributed environment.

Instead of ignoring the issues of "remoteness" of component, it must be considered in the architectural design of the system. This ensures that interfaces for communicating between objects are designed, resulting in acceptable performance measures.

2.6.2 Failures

Network systems fail in ways stand-alone systems cannot: routers go down, ethernet cables are pulled out of machines, switches are reset, and key servers can crash. These failures may be short-lived or may cause long outages. Code must be written that can handle these types of failures. The difficulty with this is that the errors may be difficult or impossible to detect. The reason for this difficulty is because distributed systems usually have no centralized entity that can be used to determine success or failure of actions.

In distributed systems, individual components can fail in ways that leave other components running, often unaware of their failure. This situation is called partial failure, and is the cause of most of the problems associated with distributed computing. Total failures are actually easier to deal with because the system ends up in a known state (i.e., up or down).

2.6.3 Concurrency and Consistency

Partial failures usually cause consistency problems. For illustration, consider the example of a shared whiteboard application that allows a set of participants to manipulate a shared drawing space. When one user makes a mark, the update must be propagated to all users in the session. If one recipient's computer becomes unreachable for a short time and the update is not processed, the participant's screen

28

may be in an inconsistent state. This would result in participants being out of sync with the rest of the workgroup. Additionally, several participants may be manipulating the shared drawing space at the same time, which causes concurrency problems. Partial failures and consistency are the real showstoppers for distributed computing. They are difficult to detect and require tremendous overhead to solve.

Early collaboration applications used floor control (e.g. one user interacting at a time) to prevent concurrency problems [17]. Later models or approaches dispensed with any type of concurrency control, relying on social protocols and global awareness to prevent conflict; however, according to Munson et al [39], social protocols and global awareness methods are not acceptable in many situations. Thus, some type of floor control mechanism is best suited to solve the concurrency problem.

To help solve the problem of concurrency and consistency, transactions can be used. Transactions ensure that complex operations will reach a "safe" state. That is, the operations either all complete, or none of them complete. This mechanism can be used to help avoid partial failures in distributed systems, to help handle concurrency and consistency issues, and to provide robustness and resiliency to network failures.

2.7 Enabling Technologies

Previous research has found the use of Sun Microsystems JavaSpaces™ to be useful in the development of a collaborative visualization framework [3, 27]. However, this research goes beyond JavaSpaces™ to look at the applicability of core technologies of Jini™ for collaboration.

2.7.1 Jini™

Jini™ technology was released to the public in early 1999. Mark Stang and Stephen Whinston of Xor Incorporated summarize Jini™ Technology in their article "Enterprise Computing with Jini™ Technology" [49]. The great power behind Jini™ is its capability to give network devices self-configuration and self-management capabilities. These capabilities enable network devices to communicate immediately on a network without any human intervention. Jini™ technology was developed as a sophisticated platform to build network-aware applications. This technology makes it possible for

users to access resources located anywhere on the network. In addition, network resources and devices can join and leave the network without manual configuration. This technology provides several advantages, such as reliability, scalability, maintenance, and security, when trying to create a network-aware application.

Reliability is the measure of how well a device or network operates when there are disturbances. Processes, services, and machines start and stop and sometimes even crash. Jini™ technology can handle these types of changes because it was designed to have resources move in and out of a network. The way it works is similar to the way machines communicate over the Internet. Between two machines there are many different paths they could take to communicate. So, if a problem occurs with one path, another path is taken. Similarly, when a server, or resource becomes unavailable on a Jini™ network, the client automatically goes searching for another server that provides the needed service, or waits for the server to come back up. This functionality is built into the Jini™ architecture and remains transparent to users.

"A system is scalable if the overhead required to add more functionality is less than the benefit the functionality provides" [49]. Adding additional Jini™ services to the network provides more choice to clients on which services they can communicate with. Adding more devices that provide the same service increases system reliability. Jini™ networks have no central control, which permits the networks to be free and manage themselves. This allows for services to be added and removed with little to no effort. The ability to dynamically discover services makes Jini™ networks fluid, meaning they change dynamically without affecting existing services.

Maintainability and administration are some other key advantages to Jini™ networks. With the ability to find the network components that make up applications, location of a source is hidden from the client. Additionally, the network is self-maintaining and does not need manual configurations. Jini™ makes fail-overs much easier to implement. For instance, a back-up server may be running, and when the primary fails, it changes its IP address and takes over the workload. This is possible because when a Jini™ server fails, a Jini™ client automatically searches for another instance of the service to finish the work.

Security is always critically important in any system, especially in the C2 realm. Jini™ software prevents unknown clients from accessing protected services. Additionally, it can prevent remote clients from even seeing services that are offered by the network. Jini™ servers maintain and transmit the client code needed to interface with a particular service. Prior to any code being moved, the Jini™ environment must satisfy the client's security policy, which defines the trusted machines and all security parameters.

Jini™ uses a service-based model to enable the sharing of resource across a network. "A service is a fundamental concept in Jini™ technology and represents an entity that users can access over a network" [49]. Services can be any entity that is abstracted through software such as printers, storage devices, and software components. These services advertise their availability and capability through lookup servers that clients use to discover and access services.

To make services available to clients several steps are followed (reference Figure 7). First, the service is bound to a *Lookup Server*. The *Service Provider* publishes its service interface or proxy code to the *Lookup Service*. The *Service Provider* may publish its service interface to any number of *Lookup Services*, thus providing redundancy if any *Lookup Server* were to fail. Second, the *Service Requester* performs a lookup to locate the *Lookup Service*. Once found, the *Service Requestor* asks the *Lookup Service* for a particular service by using a unique name. If the requested service is registered with the *Lookup Service*, it returns the proxy code, which was stored in the *Lookup Service*. Otherwise, it returns no match for the requested service. With the proxy code now accessible to the *Service Requestor*, it now knows how to communicate with the *Service Provider*. The last step is simply the communications between the *Service Requestor* and the *Service Provider* via the proxy.
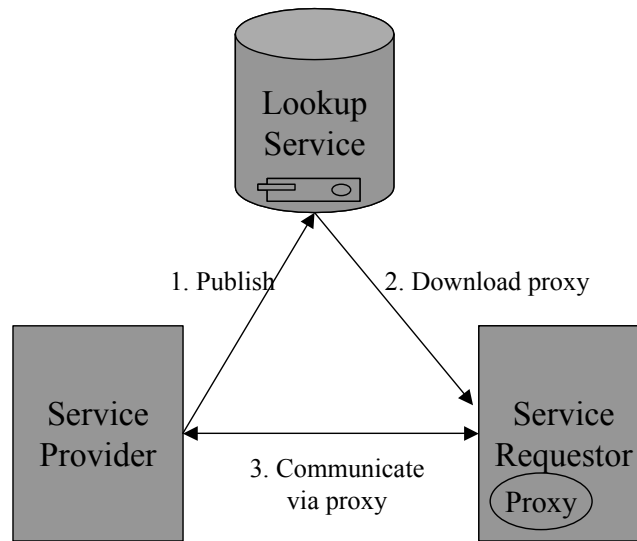
Figure 7. Jini™ Service Model [55]

The Jini™ service-based model is simple and provides great flexibility by allowing clients access to capabilities across a network. To implement this model, Jini™ technology uses five key concepts. These concepts, as described by W. Keith Edward, author of Core Jini™ [16], are: discovery, lookup, leasing, remote events, and transactions. These concepts provide the ability for Jini™ to support spontaneously created, self-healing communities of network services.

2.7.1.1 Discovery

Discovery is the process by which services find and join communities to advertise their availability. When a Jini™-aware application or service wishes to make itself available to users, it goes through the process of joining a community. It first must perform a discovery to find the lookup services and then joins them by using operations on the lookup service references that are returned by the discovery process. Once the service joins a lookup service, the service become available to use by other services and applications.

2.7.1.2 Lookup

Lookup fulfills the role of a directory service and provides the resources for searching and finding known services. Thus, it keeps track of all services that have joined a Jini™ community. It also controls how code is transmitted to clients wishing to use a particular service. The sections of code that are downloaded to clients are called the proxies and they define the interfaces to the services the client use.

2.7.1.3 Leasing

Leasing is the technique that provides the self-healing characteristics of Jini™. It will ensure that a community will recover from a failure of a key service. Access is granted to a service to exist on the lookup server for a fixed period of time. If that service does not attempt to renew its lease, the service will be dropped from the Jini™ community, thus eliminating the references to stale resources on the network.

2.7.1.4 Remote Events

Remote events are the mechanism used by Jini™ to allow services to notify other services of changes. Since lookup is a service, it can notify other interested parties of changes made to the available services.

2.7.1.5 Transactions

Transactions are the mechanism used by Jini™ to ensure that computations that require more than one service terminate processing in a "safe" state. In other words, the group of commands or computations must be completely executed together as a group for the result to be considered valid. This provides a way to "cluster" operations together to give the appearance of all the commands executing as a single command. Thus, the user can be sure that the sequence of actions completed successfully, or none of them completed, thus maintaining the system in a safe configuration.

There are three main parties involved in Jini™ transactions: the transaction manager, the transaction participant, and the transaction client. The transaction manager is just another Jini™ service

that implements the two-phase commit protocol [41] when requested. The two-phase commit protocol is a well-known database protocol that ensures operations either commit or abort. The transaction manager maintains a list of all participants for each transaction and sends messages to participants telling them to move to a different stage. Essentially the transaction manager is only in charge of ensuring that the two-phase protocol is followed.

The second party is the transaction participant. It is essentially a program that performs operations that are grouped together in a transaction. Participants must implement the *TransactionParticpant* interface.

The last party is the transaction client. It is the entity that initiates the entire process. It does not need to implement any interface. Its responsibilities are to instantiate *Transaction* objects, and ensure the transaction either commits or aborts.

2.7.2 JavaSpaces™

JavaSpaces™ is a service built on top of the Jini™ architecture that creates and maintains a database of Java objects. This mechanism facilitates group communication or collaboration through the sharing of Java objects.

JavaSpaces™ uses the standard Jini™ lookup to store and retrieve objects. The standard lookup uses the *Entry* interface to facilitate the searching for objects. The Jini™ documentation provides the following information about the *Entry* interface [55]. The *Entry* interface is the supertype of all entities that can be stored in a Jini™ Lookup service. Specific Entry objects implement the *Entry* interface and may have any number of methods or constructors. Each field in an Entry object must be a public reference object type. Each field is serialized separately, so references between two fields of an entry will not be reconstituted to be shared references, but instead to separate copies of the original object. To illustrate this, Figure 8 provides a simple example of an Entry object that has two attributes pointing to the same object prior to serialization. After serialization, the attributes now have two separate copies of the original object.

34
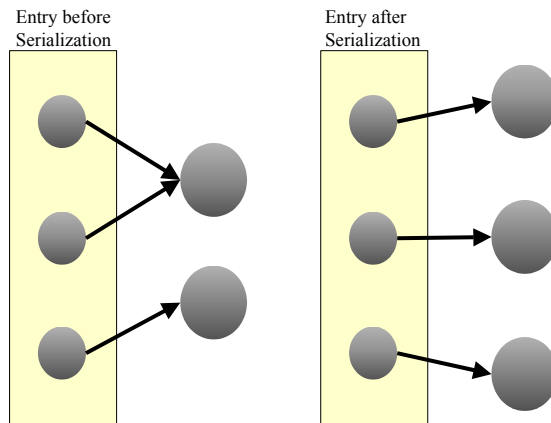
Figure 8. Entry Before/After Serialization [16]

Unlike relational databases, neither primary keys nor tables are needed to store or search for objects. Objects can be located by performing a lookup on names or any set of attributes of the object. Typical relation databases use the CRUD (create, read, update, and delete) paradigm to operate on data. Mark Stang, et al [49] showed how JavaSpaces™ uses three operations to map to the CRUD model: *write*, *read*, and *take*. *Write* performs the create and update functions, *read* maps to the read function, and *take* combines read and delete functions.

The *take* operation is used when a user wants to lock an object in the database. This actually removes the object from the space and makes it unavailable to other users. The *take* and *read* operations block until a matching object is available in the JavaSpace; however, a timeout parameter can be set to curtail the amount of time the operations wait. If the service dies after taking or locking an object, the object could be lost. To prevent this, JavaSpaces™ uses the transaction mechanism whenever a *take* is performed. Thus, if anything happens to the service, the object will be restored to its original state prior to the *take* action.

The use of JavaSpaces™ typically follows the following simple steps. First, applications or classes use the *write* method to place objects to the JavaSpace; they also remove them from the JavaSpace by using the *take* operation. Upon completion of the processing on the object, the object is written back to

the space using the *write* operation. Applications then use the *read* operation to access needed objects. Figure 9 depicts the JavaSpaces™ operations in actions.
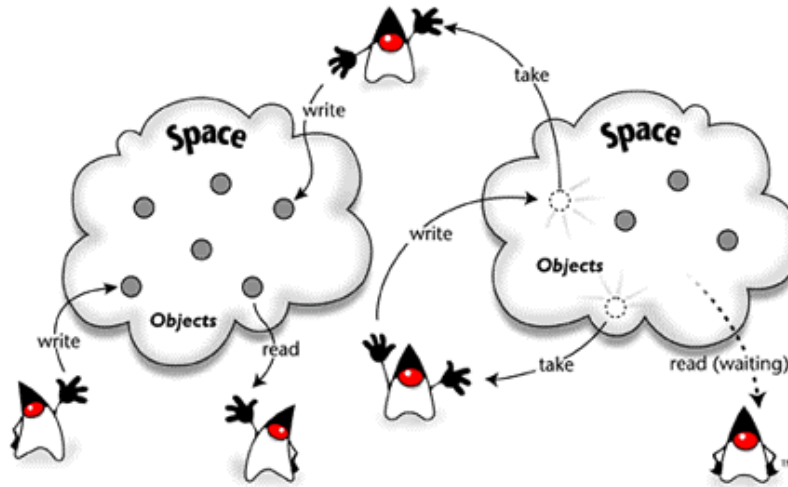
Figure 9. JavaSpaces™ Operations [19]

There are additional operations that are defined by the JavaSpaces™ interface, which are: *readIfExists*, *takeIfExists*, and *notify*. The first two, *readIfExists*, and *takeIfExist*, are exactly like their counterparts *read*, and *take* with the following exception: they will return a null if a matching entry does not exist and blocking in these calls are done only if necessary to wait for transactional states to settle.

The *notify* operation provides the mechanism for the JavaSpaces™ to notify a listener if an object matching a template has been written to the space. "The method takes an *Entry* as a template that will be matched against future writes to the JavaSpace. If a new *Entry* is written that matches the template, an event will be sent to the listener specified in the *notify* call" [16].

JavaSpaces™ Principles, Patterns and Practice [19] describes some useful properties of JavaSpaces™ that makes it a good candidate for a collaborative distributed system. These properties are summarized here:

- Spaces are shared by allowing multiple processes to interact with a JavaSpace at the same time. The concurrency issues associated with shared access are handled internally by the

36

JavaSpaces™ API, eliminating the need to write extensive concurrency control routines to handle these issues.

- Spaces are persistent, meaning that objects can be reliably stored in a JavaSpace. If an object is stored in a JavaSpace, it will remain there indefinitely until it is removed. Leases can be requested to indicate the duration of storage for the object. When the lease expires, the object is removed from the JavaSpace.

- Spaces are associative by enabling programmers the capability of locating objects in a JavaSpace with knowing the name or the location of the object. Objects are found using template objects that contains desired values/properties to search the JavaSpace. The JavaSpace matches these templates with objects in the JavaSpace and returns them. This type of associative lookup gives the programmers great flexibility by enabling them to write programs without knowledge of names or locations of other components of the application.

- Spaces allow transactions by using the Jini™ transaction service to guarantee that operations are atomic (that is, either all of the operations are applied, or none of them are applied). Transactions are essential in detecting and handling partial failure.

- Spaces allow the exchange of executable content. Objects in a JavaSpace are passive data, meaning they cannot be modified or invoked inside the JavaSpace. However, programmers can easily take an object from the JavaSpace, create a local copy and modify its public fields, as well as invoke its methods.

2.8 Related Research

Many approaches for collaboration frameworks have been attempted. Throughout time the frameworks have become more robust and provide more functionality and better performance.

2.8.1 NCSA Habanero

The National Center for Supercomputer Applications (NCSA) developed a collaborative framework called Habanero [7]. This framework is written in Java and operates by sharing actions or semantic events to multiple copies of Java applets or applications. These applications are replicated across clients that are connected to a particular session. Habanero ensures that all clients see the actions in the same order, which results in all of the applications appearing the same on each client. Programmers are given the flexibility to define what events are considered to be state changing and thus create actions to be sent to all clients.

Habanero also provides a general floor control or session controlling object, called an arbitrator. The arbitrator controls which events can be performed at a given time. The student-teacher arbitrator is a good example. This arbitrator only allows the privileged person (the teacher) to initiate actions. Thus, all the students become observers and only see the actions performed by the privileged user. Another example is the turn-taking arbitrator, which requires events to be initiated in order by each participant. Developers can also define their own arbitrators. Currently these arbitrators make TCP/IP socket connections to each client, which does not scale very well when there are a large number of clients.

Habanero also provides a collaborative environment to support the session management process. This environment allows users to create, join, leave and browse sessions. In addition, Habanero provides a suite of tools, e.g., Chat, Whiteboard, and others that are used to support collaboration between individuals [6]. A limitation to Habanero is that it requires applications to be predefined as shared tools and does not allow dynamic addition of applications [58]. Additionally, it is not very robust in being able to handle system failures.

2.8.2 DISCIPLE Framework

Ivan Marsic, et al developed a component-based framework centered on the JavaBeans™ architecture [32, 33]. The main characteristic of DISCIPLE (DIstributed System for Collaborative

Information Processing and LEarning) is a three-tiered architecture: presentation, application logic, and storage [58].

The reason for focusing on JavaBeans™ is that JavaBeans™ represents a strong software industry trend for standardizing software development through components. Components enable the speedy development of software using third party software components. In addition, JavaBeans™ help maximize the ability to decouple the communication and group aspects of collaboration from the application itself [33].

Like Habanero, DISCIPLE is also based on the replicated architecture of groupware. Therefore, each client runs a copy of the collaboration client and contains a copy of the application that is being used to collaborate. Thus, each application is kept in synch by the framework passing events to each application to make them appear to be in the same state [58].

DISCIPLE provides components that manage collaborative functions, such as concurrency control of simultaneous activities, degree of sharing of the application (coupling), and degree of group awareness within the environment [58]. Collaboration takes place in sessions, which are virtual rooms that enable users to work together. In addition, this framework has several tools that can be used to support collaboration: whiteboard, collaborative mapping, speech signal acquisition and processing, and image analysis tools. However, this framework does not handle system failures, nor does it provide flexibility to application developers to customize components in the collaborative framework.

2.8.3 COAST

The COAST (COoperative Application Systems Toolkit) provides developers with an architecture to support cooperative applications [47]. Like Habanero and DISCIPLE, COAST also uses a replicated architecture approach. Each application operates on exactly one document; thus, the data is fully replicated to each client application.

Users access the shared document via a session object. "Session objects provide group awareness and specific coupling of shared document aspects between concurrent users" [47]. The coupling can be used to implement specific cooperative modes (e.g., private, loose, tight). Users interact with the shared documents through views (visualizing the document in a windows) and controllers (processing user input from the window). Thus each view and controller object accesses the shared object through the session object. This concept is the well-known MVC (Model/View/Controller) [31] programming design pattern. This pattern decouples the functionality of the model, view, and controller to increase flexibility and reuse. This approach allows collaborating users to have multiple views of the same underlying data model [20].

COAST explicitly addresses the problems of object sharing, session management, and view updating. The session manager, transaction manager, replication manager, view, and control objects perform these functions; however, this design does not support an asynchronous mode of sharing objects. In addition, the fully replicated approach limits the size of the shareable objects.

2.8.4 ColVis

Sean Butler developed a framework called ColVis (Collaborative Visualization) as a demonstration of concept to demonstrate the feasibility of JavaSpaces™/Jini™ in a collaborative environment [3, 27]. The ColVis "framework provides communications management and message support and well-defined Java class interfaces for integrating visualization components" [27]. This framework does demonstrate that JavaSpaces™ can be used to share Java objects; however, the implementation is limited because the Java objects are not truly shared.

The framework uses a message object to pass messages to the JavaSpace, which is then accessed by each client. By taking this approach the developer must embed code that processes the messages in each new collaborative application. The framework handles the sending of the messages, but the developer is required to implement code that will pack and unpack messages to create the desired collaborative action.

Another limitation of this framework is the coupling of the communications and the user view and interactions. They are tied together in a lower level visualization components, and do not lend themselves well to maintainability and flexibility.

2.9 Background Summary

In summary, the visionary documents provide the background for why information superiority is so important in today's military environment. Obtaining information is of no use unless knowledge can be extracted from it. Collaboration and information visualization are important elements that help information become knowledge. In the command and control community, this knowledge will influence decisions. Well-informed decisions will hopefully minimize casualties and maximize military objectives.

Several distributed systems concepts are presented to help motivate the challenges that are associated with building a collaborative framework. The use of Jini™ and JavaSpaces™ technology help overcome these challenges by providing developers with mechanisms that internally handle the distributed concerns. Additionally, these technologies enable the sharing of Java objects throughout a shared workspace.

III. Methodology and Design

This chapter discusses the methodology and design used to address the objectives of this collaborative research. It describes the success criteria for the research and outlines the architecture used in the design of the collaborative framework.

As stated in Chapter I, the goals of this research are to create a generic framework that supports:

- Shared interaction – the collaboration between geographically separated users interacting with data and visual representations to accomplish tasks. The characteristics used to measure this goal are: shared state, data consistency, communication, coordination of action, monitoring, and protection.

- Visual sharing – remote users collaborate at a higher level of abstraction than the data through the sharing of visual objects. The criteria established to measure the effectiveness of this goal are visual consistency and visual collaboration.

- Remote code access –users access remote data and applications without the need for previous installation. Dynamic loading is used to measure the effectiveness of the framework in meeting this goal.

- Easy tool integration – flexibility to easily integrate any Java tool and make it collaborative. To measure this goal the following criteria are used: generality, automation, and coupling.

- Facilitate software development – the purpose of a framework is to ease the burden of developers in developing software. The following characteristics are defined to measure the capability of the framework: scalability, robustness, standard language support, and flexibility.

These overarching goals are used to define and guide decisions made during the framework development process. The choices made in developing this collaborative framework require the addressing of complex issues such as: system architecture, distributed system concerns, and global awareness.

3.1 Collaborative Framework

Most collaborative applications today have code that is specifically written to provide the collaborative functionality desired in the application. This paradigm couples the application code to the collaborative framework, thus forcing every new application to implement similar collaborative constructs. This approach seems to be effective, but creates excessive burdens on application developers to add collaborative features to each new application.

This research uses a generic approach to try and build a framework that would take any Java™ application and make it collaborative in nature. This generic approach decouples the application from the collaborative software, thus eliminating the need for the application to implement distributed system concepts and making it easier for developers to create new applications.

3.1.1 Criteria

A review of related work [5, 18, 20, 22, 23, 32, 59] has revealed certain characteristics that are desirable in a collaborative framework. These criteria encompass characteristics for a good software framework, as well as characteristics needed to effectively collaborate with geographically separated individuals.

- *Generality*: By producing a general solution the application of the framework can benefit more people and be used across many domains. This characteristic allows the use of a single framework to develop domain-independent collaborative applications, providing flexibility and reuse. Using a single framework to support many different domains eliminates the need to learn a new system to create a collaborative system in other domains.

- *Automation*: Automation is the spontaneous support of making an application collaborative with minimal intervention from the developer. The level of automation can be measured by the "developer's effort [required] to achieve multi-user behavior with respect to a single-user case" [46].

- *Shared State*: All clients in the collaborative environment must remain synchronized. A common approach to maintain synchronization is to share a common state model. This synchronized state may be maintained in any number of ways; however, since the goal is to allow for interaction with both data and visual representations, a mechanism for sharing of objects is the desired approach to be integrated into the framework. This approach allows for both data and visualization to remain synchronized because each client is using the same objects.

- *Consistency*: The framework must support both internal data consistency as well as display consistency. In other words, the framework must ensure the state of the shared objects and their visualization remains the same. This is essential for effective collaboration to take place.

- *Scalability*: The approach should be scalable with respect to the number of users and the size of the shared applications that are to be used. Increasing requirements should be met by components that are flexible and able to handle a greater load (e.g. by adding more servers). The upper bound is not constrained, however, the focus of this framework is to facilitate the sharing of ideas to aid in decision-making. Too many people involved in the collaboration makes the ability to come to any consensus difficult.

- *Robustness*: Along with scalability, the framework should be able to handle system failures. This can be best described as self-healing. When servers go, does the system crash, or is the framework able to account for the change and continue the collaboration activities?

- *Communication*: A collaborative environment must be able to support both explicit and consequential communications. Communication is the main focus of a collaboration framework, because it enables individuals to work together to accomplish a task. All actions must be handled

and processed by the collaborative framework in a synchronized fashion. That is, each event of action must be processed in an ordered manner according to when the action took place.

- *Dynamic Loading*: Not all users may have every instance or correct version of an application, thus creating an inconsistency in the collaborative environment. To prevent this problem, the framework should support the ability to dynamically load classes from a common repository.

- *Coupling*: Coupling is the measure of how much the application is joined to the collaborative framework. The desired approach is to create a framework that is decoupled as much as possible from the applications. This creates a desired separation and independence between the framework and the shared applications.

- *Standard Language Support*: The framework should avoid the use of additional support from languages (i.e. specialty compilers, interpreters). The framework should not be bound to a particular language or platform, thus enabling independent modification.

- *Visual Collaboration*: One of the goals of the framework is to enable users to collaborate at higher level of abstraction from that of sharing data. To measure this requirement, we ask the question: how well does the framework support sharing of visual entities?

- *Flexibility*: Does the framework provide the capability for programmers to devise their own extensions and to customize system behavior? The framework should enable developers to customize components easily with minimal side effects.

- *Coordination of Action*: To avoid conflicts within a shared environment, participants must take turns. In addition, some tasks may require a sequence of events to take place in a certain order. Thus, it is imperative that collaborants can organize their actions within the shared workspace so conflicts do not arise.

- *Monitoring*: Almost all other mechanisms for collaboration rely on the ability to monitor and collect information about others in the workspace. Workspace or group awareness information,

such as who is in the workspace, where they are working, and what they are doing, is the primary

information needed to enable collaborants to make progress on tasks and effectively work

together.

- *Protection*: One obvious potential problem with a shared workspace is the inadvertent altering of

  work by another in the workspace. Thus, people must be courteous of others' work and not

  inadvertently destroy someone else's work. In addition, mechanisms for locking different

  elements or regions of the shared workspace with passwords may be necessary to provide

  adequate protection to a user's work.

3.1.2 Visual Collaboration

Collaboration happens at many different levels. Section 2.4.2 discusses collaboration at the data,

data abstraction, view abstraction and view levels. An important objective of this collaborative framework

is to support collaboration at a more abstract level than that of the data level. Visual sharing or view

abstraction is a desired feature within the framework.

Figure 10 shows three levels of collaboration that can be used to establish communication between

two or more clients: data, interactive events and visual. Data collaboration is the sharing of information via

a file or raw data. The ever-increasing popularity and use of XML is making this approach more feasible

and preferable to application developers. Interactive event sharing is the propagation of event messages to

all users. This is the most commonly used method today. This may be due to the minimal bandwidth

required to send event messages to clients participating in a collaborative group. Visual collaboration is the

sharing of visual objects within the collaborative environment. Visual collaboration is the approach taken

in this research due to its appeal to minimize the complexity of the framework and enhance flexibility.
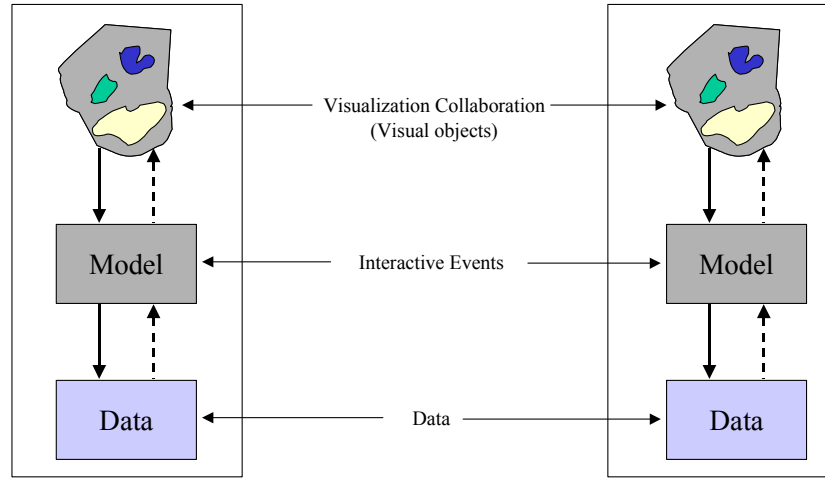
Figure 10. Visual Collaboration Abstraction Diagram

3.2 System Architecture/Design

In this research, the Collaborative Visualization Environment (COVE) framework is designed to provide a separation of semantics of applications from semantics of the collaboration and distributed computing. This enables applications to collaborate having no knowledge of the underlying collaborative mechanisms. This approach is similar to the DISCIPLE [33] approach, with a dramatic shift in the collaborative mechanisms. DISCIPLE uses a replicated architecture using an event propagation approach while COVE uses a centralized shared object approach. The primary advantage of using the shared object approach is gained by eliminating the need for extensive concurrency and consistency control mechanisms.

3.2.1 Architecture Choices

The two most common architectures used in collaborative frameworks are: centralized and replicated. These architectures have subtle difference that can impact the run-time performance, functionality, and scalability of resulting collaborative applications.

In the replicated architecture, [refer to Figure 11] exact copies or replicas of the application being shared must be installed and maintained on each host. The application on each host handles the user interaction locally. Any changes made to the application state are broadcast to all other replicas to maintain the consistency of the data and the user views. The major disadvantages of replicated architecture are the difficulties in keeping the data and user views consistent and in synchronization. Since the applications

47

are managed locally, two users may interact with the application at the same time instant in different ways, thus resulting in inconsistency. Complex synchronization techniques are needed to maintain harmony among the replicas and all user views. [29]

Synchronization signals

| Application (host 1) | Application (host 2) | Application (host 3) | . . . . | Application (host n) |

| Screen (host 1) | Screen (host 2) | Screen (host 3) | . . . . | Screen (host n) |
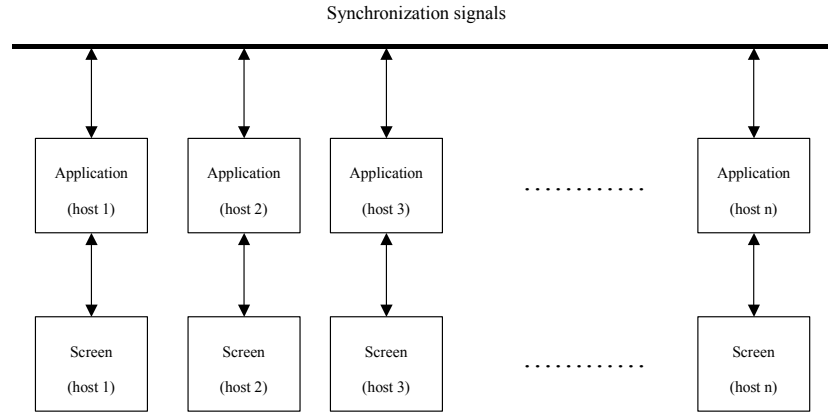
Figure 11. Replication Architecture [29]

The strength of the replicated architecture lies in its user responsiveness and robustness. The user interacts with local copies of objects thus increasing the responsiveness and performance of the applications. Robustness is increased because a single point of failure does not exist.

The centralized architecture as described by [5], "executes a single instance of the collaborative application and maintains one copy of the shared data, allowing the single application instance to process multi-Input/Output (I/O) and support collaborative functions." The main advantage of the centralized architecture is the absence of synchronization, as clients communicated via sequential I/O with a single copy of the shared data. This approach also simplifies other replication problems (e.g. concurrency duplication tasks). The primary disadvantage to the centralized architecture is the increase response lag. Since all the objects are shared over a network, latency and bandwidth limitations affect the performance of the system. Figure 12 shows how the application is run on a single application with other hosts accessing the single application.
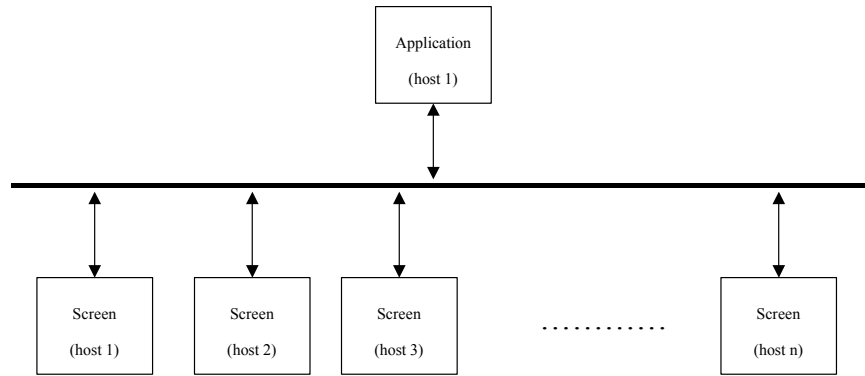
Figure 12. Centralized Architecture [29]

The design chosen for this research is a combination of the two architectures to take advantage of the strengths of each. The centralized architecture reduces synchronization issues while the replication architecture provides quicker response time. Thus, the resulting architecture is a centralized repository of shared visual objects with a local replicated application on each host (refer to Figure 13). Each host gets access to a centralized shared object set providing consistency and synchronization controls. Each local host also maintains a local copy of each collaborative application to increase responsiveness and performance.
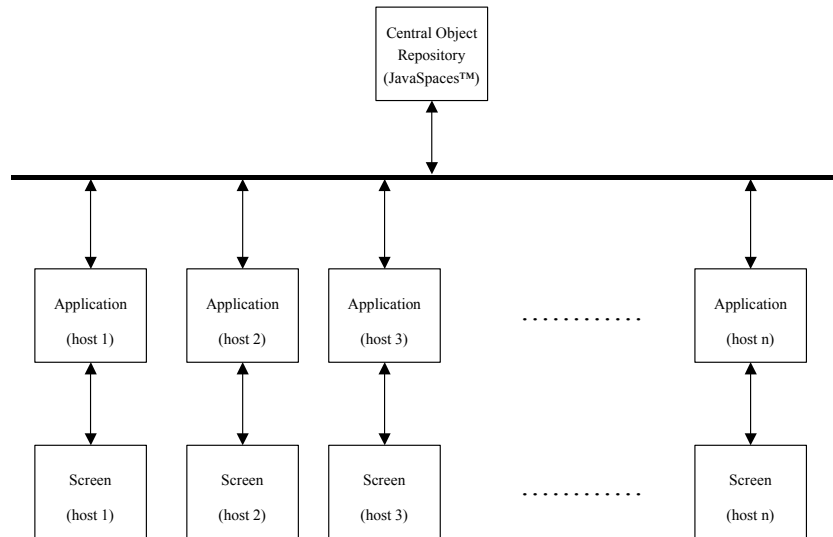
Figure 13. COVE Framework Architecture

3.2.2 Session Service Architecture

To create a framework that decouples the collaborative framework from the application the concept of a service is used. A service is an entity that provides a set of well-defined operations that can be accessed anywhere on the network. A service is comprised of a *service provider* and a *service proxy*. The service provider publishes the service proxy to the network so clients can have access to the capability it facilitates. Clients fetch the service proxy from the network and then execute it on their local Java Virtual Machine to interface with the service provider.

The COVE framework uses this concept by establishing a Session Service running on top of the Jini™ network. Jini™ technology [55] is an innovative and usable technology for building reliable, fault-tolerant distributed applications. Jini™ provides an infrastructure that allows clients to find services independent of both party's location. Figure 14 illustrates how the Session Service sit on top of the Jini™ network and how client applications interact with the Session Service via the Service Interface or proxy. Clients interact with other clients through a generic set of operations supplied in the Session Service. This simple design decouples the collaborative elements of the framework from the client application by keeping the collaborative elements strictly in the service. Clients with access to the service interface need never know (nor care) about underlying network classes or interfaces that support collaboration.
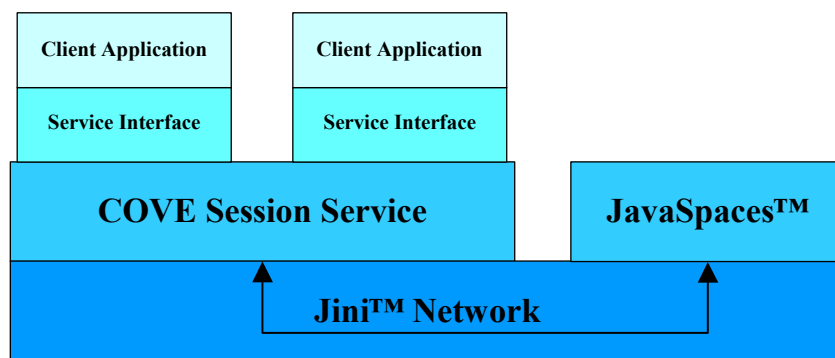
Figure 14. COVE Layered Architecture

The Session Service also has the responsibility to provide the sole interface to the shared object store. Once again, this simplifies the client application by interacting with the session service interface.

Additionally, this design choice enables clients to dynamically load the proxy or interface from the network to gain access to the service. This is very powerful in that it allows clients anywhere on the network to use a service without any explicit driver or software installation.

3.2.3 Object Sharing Via JavaSpaces™

In this research a shared object space establishes the centralized architecture that is used as the collaboration mechanism. As discussed by Ceglar and Calder [5], "object sharing reduces the need for structural redundancy required to support the view of multiple roles of data, and reduces the number of instances of objects." The object sharing service provides mechanisms for keeping the shared object space in a consistent state, thus enabling participants to monitor and plan actions to accomplish tasks. This approach reduces the number of dependencies that must be maintained in order to maintain consistency.

The shared object approach provides robustness by allowing a global access point to share an object anywhere across a network. Clients can exchange executable content by passing instantiated objects from one client to another.

Redundancy can be accomplished by running multiple synchronized JavaSpace services, such that when one JavaSpace goes down the other one takes over. This capability can avoid a potential single point of failure.

The shared object space utilizes JavaSpaces™ technology as the storage and sharing mechanism in the collaborative environment. JavaSpaces™ provides concurrency and consistency controls for the shared objects, thus eliminating the need for additional complex synchronization and concurrency control algorithms in the framework.

3.2.3.1 Concurrency Control

COVE's concurrency control model is based on a first come, first served basis. This type of floor control enables collaborants to opportunistically aid each other in accomplishing tasks by coordinating actions in a real-time fashion. When users desire to manipulate an application in the shared environment,

they first make a change to their local copy of the shared object. The framework takes the original object from the shared space and replaces it with the local changed object. The JavaSpace then notifies each client that a change has occurred, which then reads the new object from the space and updates its screen. The synchronization process only locks the shared object for the duration required to replace it with the changed one, thus collaborants cannot lock an object indefinitely.

To design this concurrency control, objects that are being manipulated are removed from the space and become inaccessible to other users. Users desiring to manipulate an object will wait until the object becomes available in the space. The removal of the object from the shared repository gives exclusive use to the participant who gets access to the object first, hence first come first serve. Upon completion of making a change to the object it is then written back to the space and propagated to all clients participating in a session to maintain consistency. The use of the centralized and replicated architecture is evident in the concurrency control mechanisms. The object repository receives and stores the changed object, but each workstation receives a copy of that object to display in their local workspaces for the benefit of user responsiveness. The workspace is discussed in greater detail in Section 3.2.4.2.

3.2.3.2 Consistency Control

Consistency is another difficulty facing designers of a collaborative framework. As discussed in Section 2.6.3, this issue can be one of the most complex issues to solve. In this research the shared object approach is used to maintain consistency in the collaborative environment. By maintaining a repository of shared objects, consistency can be maintained, because only one set of objects are used for collaboration. When objects are changed, the changes are reflected in the shared repository. Thus, any participant in the collaborative environment will have to access this repository to get the current state of the objects.

3.2.3.3 JavaBeans™

The shared object repository must store objects and make them available to clients across a network. To develop a framework that supports every type of application and programming construct

without any constraints would be an arduous task if not impossible. Thus to limit the set of applications, the JavaBeans™ [52] component architecture is used. "A Java Bean is a reusable software component that can be manipulated visually in a builder tool" [52]. However, this is a very vague definition. For this research a bean is considered an instantiated group of Java classes that follow the JavaBeans™ specification. The restriction to JavaBeans™ ensures that each application implements the *Serializable* interface, thus enabling the storage and movement of objects across a network. The *Serializable* interface is used to identify objects that can be converted into a stream of bytes then later be put back together into an identical object.

The JavaBeans™ component architecture is a platform-neutral architecture for the Java application environment. It is the ideal choice for developing network-aware solutions for heterogeneous hardware and operating system environments for the following reasons:

- Common development standard: JavaBeans™ follow common standard design patterns. These patterns define a standard way beans are coded and enable the mechanism of introspection.

- Serialization/Persistence: Serialization and bean persistence are major considerations for the sharing of objects. These mechanisms enable objects to be stored for indefinite periods of time and to be sent to other clients across a network. This capability is critical to the approach taken in this research.

- Introspection: JavaBeans™ conform to a well-defined design pattern, which is used to obtain information about the bean. The Java Reflection API is used to look inside the bean and extract information about its composition. This feature is used to ensure an application meets the constraints that will work in the COVE framework.

- Event Model: The JavaBeans™ architecture uses a delegation-based event model. An event is something of importance that happens at a specific point in time (e.g. user clicks a mouse button). The event model is used to identify changes to application beans within the

53

collaborative environment. This mechanism is critical in signaling the framework to send distributed events that are used to maintain consistency.

3.2.3.4 Session Manager

The most important shared object that is stored in the object repository is the *SessionManager*. Session management is a key element of the framework that keeps track of what sessions and users are in the collaborative environment. The *SessionManager* maintains the current state of the environment and is responsible to ensure that the state of all users is kept consistent, regardless of when a user enters a session.

3.2.4 Replicated Client Architecture

At the end of Section 3.2.1, the design solution introduces a combined approach of using a centralized and replication architecture. The centralized architecture of the framework provides the backbone to facilitate collaboration through the uses of a service and a shared repository of objects. How the user interacts with the backbone is just as important.

The user interface uses a replicated architecture to display the applications in the collaborative framework. This approach is useful in providing greater responsiveness to users. Each client uses a local desktop to display the current state of the session manager. Workspaces show the current state of the session by displaying applications that are being used. The state of these sessions and applications are maintained by the object store; however, each client has a local copy of the application that is updated when changes are made to the shared object. Each client maintains a copy of the global state and notifies the object store of changes that are made to its local copies. This notification is important, because it forms the basis for the collaborative mechanism used in the framework.

The desktop and workspace must support the concept of group awareness in order to enable participants to monitor and provide assistance to each other. Ceglar and Calder [5] stated, "Global awareness facilitates multi-user coordination within a collaborative application by providing users with information regarding other collaborators, allowing individual users to maintain a global perspective of the

collaboration." The implementation of awareness enables coordination of actions between users, reducing conflicts and duplication of work.

According to Koch et al [30] there are several common mechanisms used to provide awareness information. Status and events are the most important mechanisms for awareness information. Status is the information collected about the collaborative participants within the collaborative environment. The events provide the real-time workspace awareness. The session manager provides status to all users in the collaborative environment, while the framework processes the events to produce collaborative real-time workspace awareness.

3.2.4.1 Desktop

The *Desktop* is the user interface used to represent the current state of the session manager, as well as provide a set of operations on which to interface with the collaborative environment. It enables users to discover what sessions (workgroups) exist, find out who is currently in a session and to join one of them. As shown in Figure 15 a simple tree-like design can be used to show hierarchy and containment within the collaborative environment. The design of the *Desktop* provides a global picture of all the users and available sessions within the environment. Thus, users are enabled to form workgroups spontaneously, in sessions, to collaborate and accomplish joint tasks.

As a user joins a session, all the objects that are in that session are shared and become available to that user. Thus, users are allowed to manipulate a group of shared objects to accomplish some desired task or operation. Figure 15 provides a snapshot view of the desktop tool.

The functions or operations that can be performed from the *Desktop* are: login to system, logout of system, create a session, remove a session, join a session, leave a session, and drop a bean. This provides users the basic functionality needed to create spontaneous collaborative sessions in which work can be accomplished.
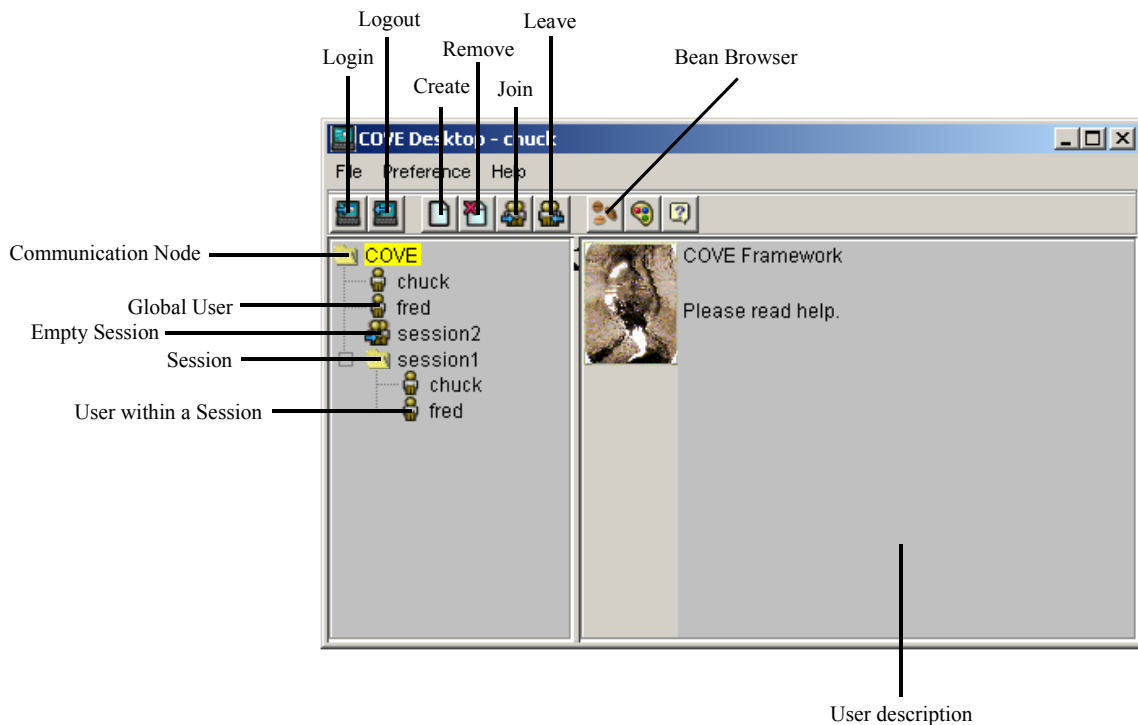
Figure 15. COVE Desktop User Interface

3.2.4.2 Workspace

The *Workspace* is a local container, which holds all the applications (JavaBeans™) that are part of a session. When a user joins a session, a workspace for that session is automatically launched and the current state of the session is displayed. Users can then add and remove beans from the workspace by loading them or closing them. The workspace provides a semi-WYSIWIS (What You See Is What I See) look and feel because all the application will act the same, but the locations of the applications are independent of each other. Figure 16 illustrates how two separate workspaces have the same applications, but the locations are maintained independent of each other.

The *Workspace* encapsulates each application (bean) in a *JInternalFrame*. By so doing, it decouples the beans location attributes from those of the global ones. This enables the bean's screen location to remain a local parameter instead of a shared attribute. Additionally, the encapsulation provides

56

a generic way to capture events to know when an object changes state. This is the trigger that synchronizes the clients with the shared repository.
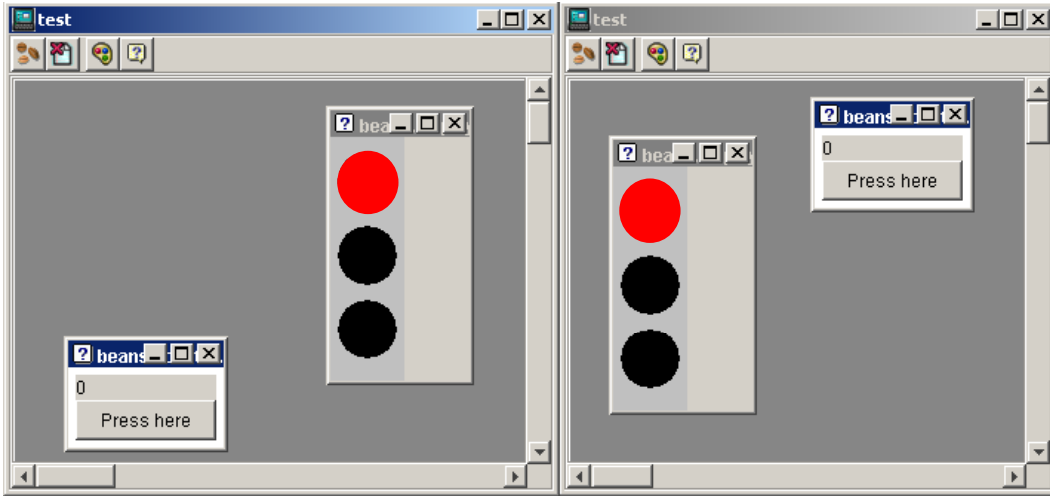
Figure 16. COVE Workspace User Interface

3.2.4.2.1 Bean Loading

One of the goals of the framework is to decouple the collaborative framework from the applications and allow code to be accessed remotely. With this in mind, there needs to be a way to link the application to the framework and enable the framework to access code remotely. To accomplish this, the concept of dropping a bean into a workspace is used.

Bean loading provides the primary mechanism to connect the framework and application together by having a bean loaded into a workspace. Each bean is encapsulated by a *BeanData* object, which is an extension of a *JInternalFrame*, to interface with the collaborative framework. The *BeanData* object performs two main functions. First, it stores the beans and second it is used to capture events to synchronize them in the collaborative framework.

Bean loading takes advantage of the ability of the Jini™ network service to load JavaBeans™ remotely. The bean loader uses introspection to gather information about the beans and then displays the available beans in the Bean Browser, refer to Figure 17. When a bean is selected and then dropped into the

workspace, the framework uses the network services to locate the class file for the bean to instantiate an object. This capability provides great robustness to the design and allows for clients to access software not previously installed.
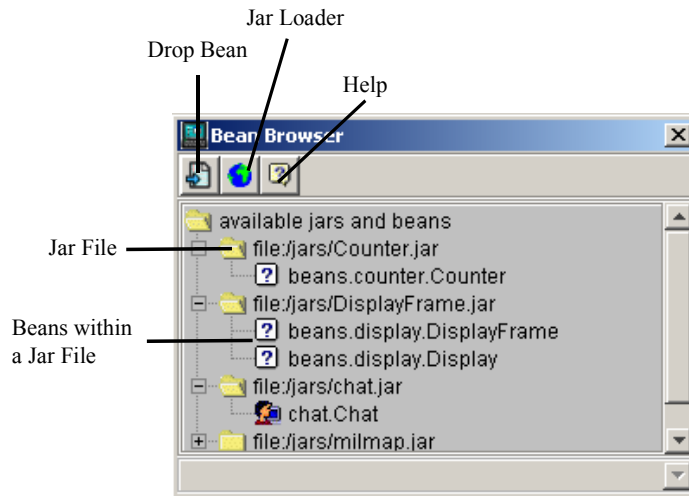
Figure 17. Bean Browser

3.2.4.2.2 Bean Synchronization

When a bean is dropped into a workspace, a glass pane is placed over the bean to be able to detect events as they occur. These events are used to trigger the object synchronization mechanism within the framework. Synchronization is maintained as objects are taken from the shared repository, manipulated, and written back to the shared space. When objects are written back to the repository, each client in the session reads the new object and updates its view. This four-step process of taking, manipulating, writing, then reading provides the mechanism for collaboration of beans in the workspace.

IV. Implementation

This chapter presents the implementation details used to create the collaborative visualization framework to meet the criteria and design defined in Chapter 3. Issues surrounding the implementation are discussed where applicable.

4.1 Session Service Implementation

The service design takes a client-server approach by providing a central place where collaborative operations can be processed. This central approach provides a common interface that each client uses to interact within the collaborative framework. The system is implemented using Jini™ services to provide the necessary remote connectivity needed to support collaboration between remote users. This technology provides the desired flexibility and ease of use desired in a framework. It also satisfies several other characteristics, defined in Section 3.1.1, such as: shared state, dynamic loading, scalability, robustness, and standard language support.

The design centers on the concept of a service provider and a service proxy providing a set of functionality to users across a network. The power of this concept is that it enables an application to download an interface or proxy from anywhere on the network to use the particular service. For example, Figure 14 illustrates how the application uses an interface to interact with the session service.

To make this concept a reality, an interface named *ISessionService* is created that defines what the session service offers. The *ISessionService* provides a set of methods used to enable collaboration to take place. The actual implementation for the service is found in the *SessionServiceProxy* class. The methods the service offers are shown in Figure 18.

As discussed in Section 3.2.2, the design must consolidate all the interaction with the shared object repository into a single service. The *SessionServiceProxy* creates the interface to the JavaSpace, by consolidating most JavaSpaces™ operations to one class. This simplifies other components in the

framework, because they need not know anything about JavaSpaces™. These components simply use the *ISessionService* interface to interact with the JavaSpace.

   ISessionService and *SessionServiceProxy* classes define the session service, while the *SessionServiceWrapper* class implements the service provider. The service provider binds the service proxy, in this case the *SessionServiceProxy*, to the Jini™ lookup server so it can be available to clients. Additionally, it maintains the instance of the *SessionServiceProxy* object that processes the requests. The service provider remains running indefinitely as a means to keep the service proxy available to users. This is necessary due to the use of the Jini™ leasing feature.

   Leases make services available for a certain period. When that time expires, the service provider must renew the lease or the service is dropped from the lookup server. This mechanism adds robustness to the network by enabling self-healing to take place. When a service crashes, the client using that service will eventually time-out and begins looking for another service to communicate with.
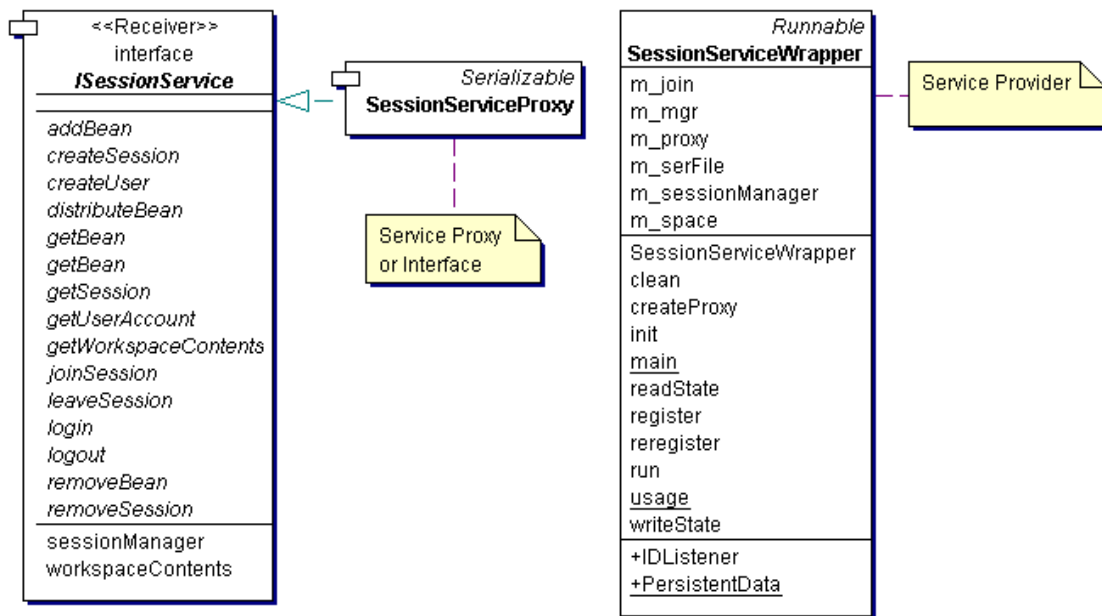
Figure 18. Session Service UML Diagram

4.2 Collaborative Framework Components

4.2.1 SessionManager Class

The *SessionManager* is used to maintain global state in the collaborative environment. Thus, to ensure that only one *SessionManager* object is instantiated within the environment and provide a global access point to the object, the singleton pattern is used.

The singleton pattern [20] is a creational design pattern used to ensure that a class has only one instance and can be accessed through a well-known access point. This pattern is very useful in cases where one and only one instance is desired, and global access is needed. The example of a print manager illustrates the need for this pattern. There may be many printers on a system, but there should be one and only one print manager and it must be globally accessible. This is due to the fact that the print manager keeps track of all printers on the network and provides information to clients concerning those printers. If there were multiple print managers, information may be lost or incomplete due to two entities trying to maintain control of the printers. To make this possible the class is made responsible for keeping track of its sole instance. The class can then ensure that no other instance can be created (by intercepting requests to create new objects). Additionally, it provides a global access point to the instance through a static method called getInstance().

Figure 19 shows a UML diagram of the *SessionManager* class. It has an attribute called *m_instance* that is used to keep track of the sole instance of the class. The visibility of the constructor is private, so no outside object can instantiate the *SessionManager* class. The well-known access point, *getInstance()* method, creates the sole instance of the *SessionManager* and returns a pointer to that instance. The *getInstance()* method is static, so it can be accessed without creating an instance of the *SessionManager*.

The implementation of this pattern is not only convenient, but also essential in ensuring that only one *SessionManager* object exists. Since the collaborative framework depends on the *SessionManager* to maintain the state of the collaborative environment, one can image what would happen if multiple

SessionManager objects were in the environment. Collaboration would be impossible, because every client could have a different view of the environment.
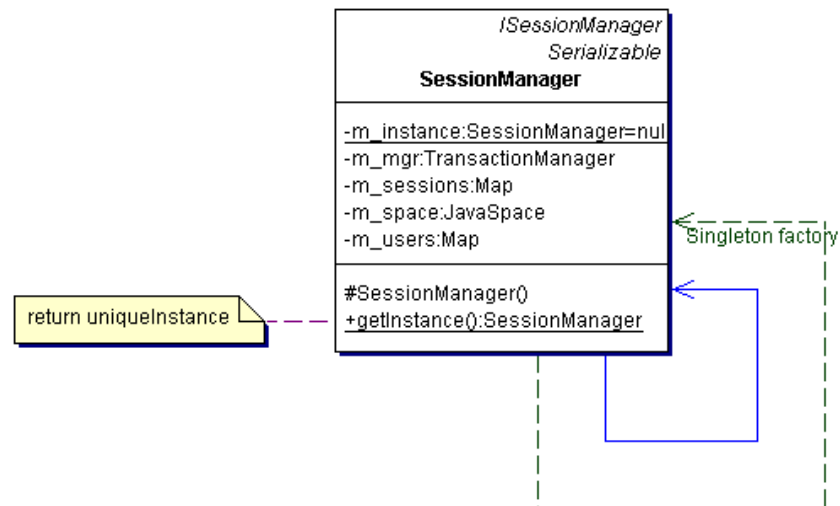
Figure 19. *SessionManager* Class Diagram

For the *SessionManager* to maintain state of the collaborative environment, two attributes are used: *m_sessions* and *m_users*. The *m_sessions* attribute maintains a list of all the sessions and the *m_users* attribute maintains a list of all the users logged into the collaborative environment. Both of these attributes use the Java *Map* interface to decouple the actual data structure from the code. This concept is known as programming to an interface, thus allowing the underlying data structure to change without affecting any other code that uses that interface. The Java *HashMap* is the data structure used for both attributes due to its simplicity and quick access time.

An important implementation decision is made on how to store this information. At first, the m_session attribute stored session objects and the m_users attribute stored user objects. This made logical sense from a single client perspective, but caused inefficiencies in the object sharing approach. Updates required the serialization and deserialization of many unnecessary objects from the JavaSpace. So to solve the problem an object relation approach was used. The *m_sessions* and *m_users* collections now only store identification tags or keys and not the actual session or user object. This still provides the objects with the

needed information but removes the aggregation constraint that caused the unnecessary serializations. This approach now enables the framework to directly search the JavaSpace for the desired objects and serialize only the objects needing to be updated.

4.2.2 Desktop Class

The desktop is broken in two main functional pieces. First, the desktop provides global awareness, by displaying information about users, and sessions currently in the collaborative environment. The information is stored in the *SessionManager* class used throughout the framework to maintain global state and provide the model needed for awareness. The *SessionManager* is the underlying data model used in the *Desktop* monitor display, shown in Figure 20. This view provides global awareness to collaborants and enables the spontaneous creation of groups to accomplish tasks.
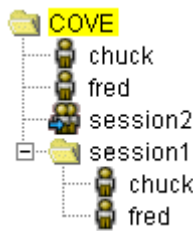
Figure 20. Desktop Monitor Display

Second, the desktop provides a set of operations with which to interface with the collaborative environment. The *Desktop* class uses the already defined and implemented Session Service to perform all these operations (e.g., Login, Logout, Create a session, Join a session). To use the Session Service the *Desktop* is responsible for finding the service and downloading the proxy code so it can use it. To do this, the *Desktop* first uses Jini's discovery mechanism to locate a lookup server; then it uses the *lookup()* method from the Jini™ *ServiceRegistrar* class to find the desired service. This method, *lookup(ServiceTemplate tmpl),* takes a service template object as input and returns the service object that matches the template or null if there is none. If there is none, the Session Service has most likely not been started or has crashed. Once the desired service is found, the *Desktop* interfaces with the service via its

proxy. Since the proxy is just another instantiation of an object, the *Desktop* keeps a reference to it, in this case the *SessionServiceProxy*. This reference is passed to any other class that needs access to the service.

Figure 21 illustrates this concept of how the Desktop uses the Session Service to perform the collaborative operations. The Graphical User Interface (GUI), in this case the *Desktop*, gains access to the *SessionServiceProxy* via the process described above. Then the GUI makes requests through method calls to the SessionServiceProxy. The proxy forwards those request to the service, which then processes them. The Session Service interfaces with the JavaSpace to manipulate and change objects based on the nature of the request. For example, if a user logs into the collaboration session, the Session Service will update the *SessionManager* with a new user ID and create a new *User* object and store it in the JavaSpace. If the method call has a return value, the service returns a value on completion of the method.
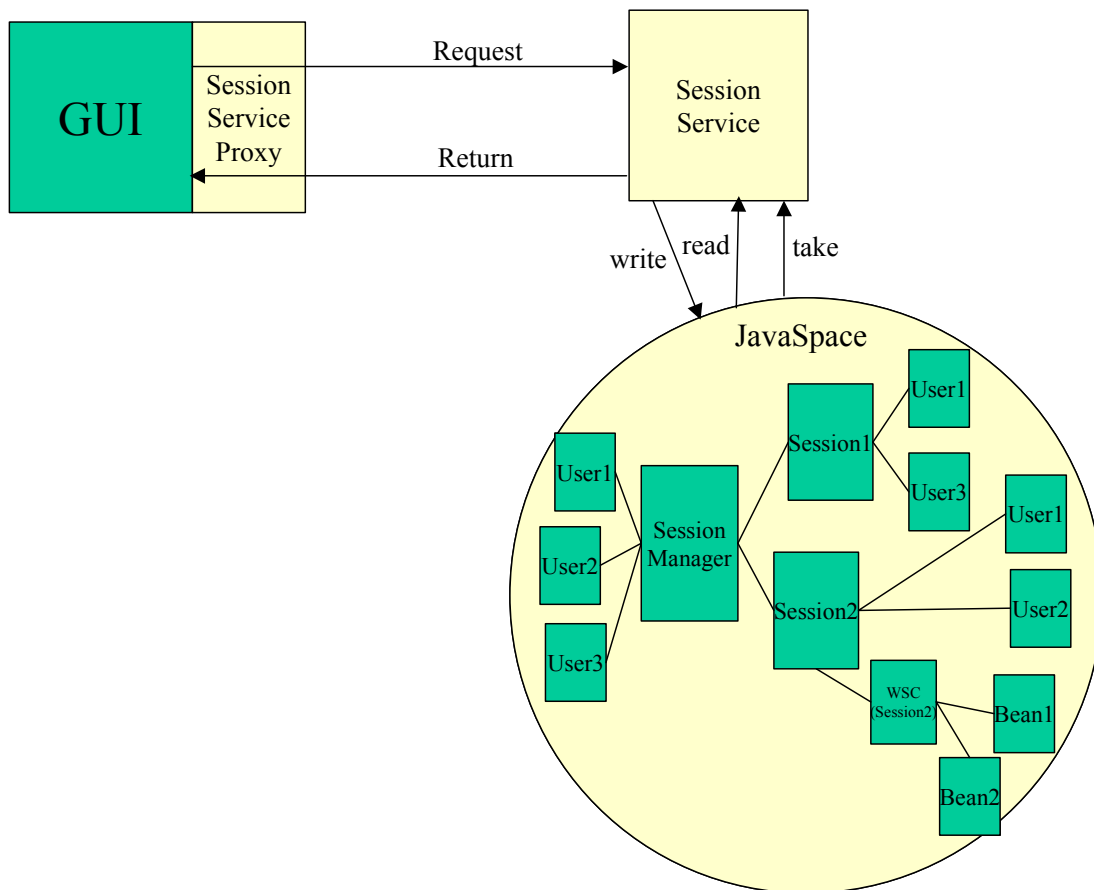
Figure 21. COVE - GUI to Service Relationship Diagram

The Session Service performs all these operations, but menus and buttons on the desktop graphical user interface are still needed to tie the desired functionality to the graphical user interface. The desired functionality (e.g., Login, Logout, Create session, Remove session, Join session, Leave session) enables collaborants to interface with the framework. To avoid placing large amounts of code in the Desktop class, yet provide the menus and buttons with the information needed to handle requests, the command pattern is used.

The command pattern [20] is an object behavioral pattern that enables applications to achieve complete decoupling between the *invoker* and the *receiver*. An *invoker* is an object that requests an operation, and a *receiver* is an object that receives the request to execute a certain operation. The key to this pattern is a *Command* interface, which declares an interface for executing operations. This interface includes an abstract *execute()* method that must be defined by concrete classes. These concrete Command classes specify a receiver-action pair by storing the receiver as an instance variable and provide different implementations of the *execute()* method to invoke the request.

Figure 22 provides an example of how the command pattern is used in the framework. Consider the case where a user is logging into the system. The *Desktop* class is the invoker because it calls the *execute()* method of the command interface through the *actionPerformed()* method of the inner class, *SessionHandle*. *SessionHandle* is a simple helper class that handles all events generated in the *Desktop* graphical user interface. The concrete command, *LoginCmd*, implements the *execute()* method of the command interface. The *LoginCmd* has the knowledge to call the appropriate receiver object's operation to perform the desired task or operations. In this case, the *execute()* method makes a call to the *login()* method of the *ISessionService* object. Here the concrete command acts as an adapter between the invoker (*Desktop*) and the receiver (*ISessionService*) objects.

By using this pattern, the menus and buttons need not know how to handle a request or operation, but instead, invoke the abstract command to execute the desired operation. The *SessionHandle* inner class actually instantiates the correct concrete command that is to be run, then calls the *execute()* method of the new command object. This simple design pattern enables the *Desktop* class to offload the specific

implementations of the commands it uses to concrete command classes. The concrete command classes have the necessary information needed to carry out the desired operation.
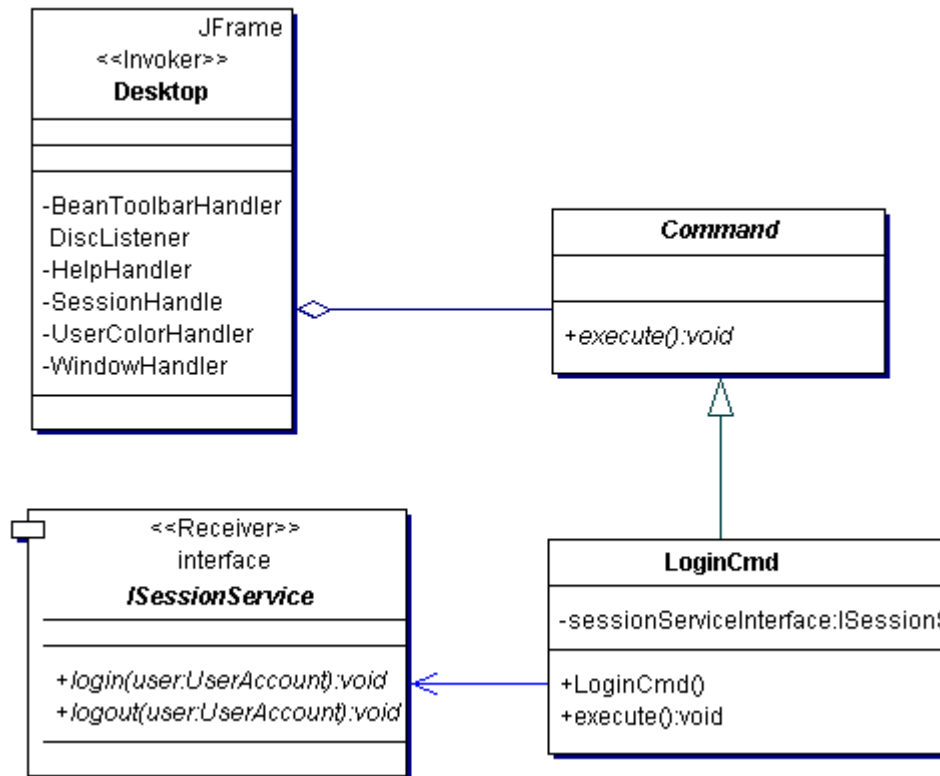
Figure 22. Desktop Command Pattern Usage

All concrete commands inherit from an abstract class called *Command*. This permits the buttons and menus of the *Desktop* to invoke all requests by using the *Command* interface; not knowing exactly which command will be executed at design time. Figure 23 shows the concrete commands used in the COVE framework. Each one of the concrete commands uses the *ISessionService* interface as the receiver of the desired action or request.

The use of this pattern decouples the buttons and menus of the *Desktop* from the *ISessionService* and allows for a more robust implementation. New commands are easy to add, because existing classes do not need to be modified. This concept encapsulates all the functionality required for a certain command and provides a central control mechanism to execute the requested action.
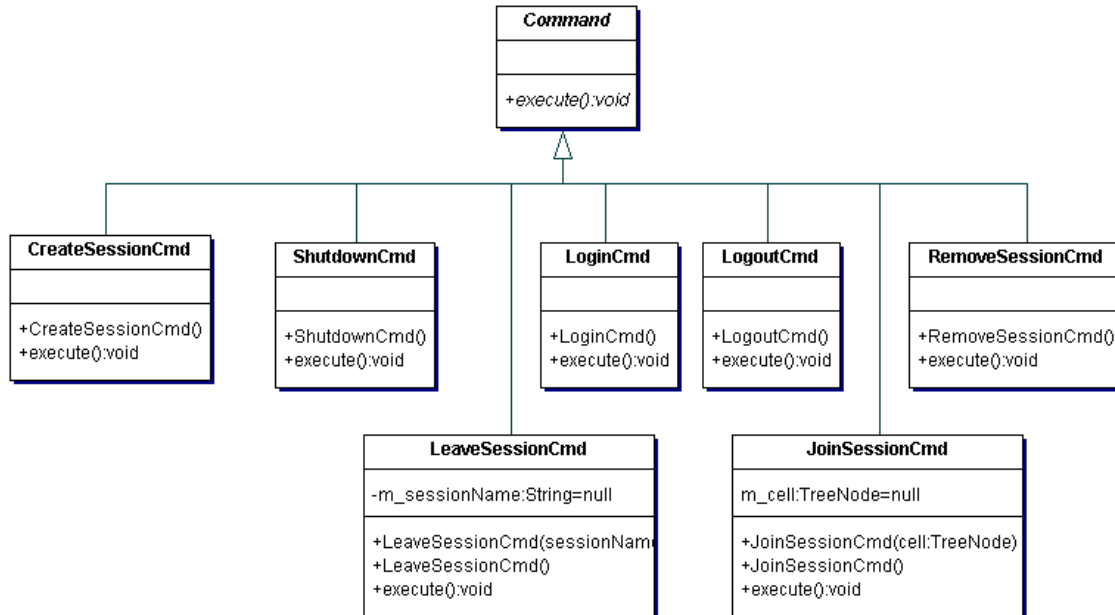
Figure 23. Command Pattern Class Diagram

4.2.3 Session Class

A session is a virtual meeting place where collaborants meet to work together. The *Session* class provides the needed data model to facilitate collaboration with members of a session. Its main purpose is the keep track of the users involved in a collaborative session and to interface with the *WorkspaceContents* to keep track of the beans within a session. The Session has the responsibility of giving the bean a unique identification number. This unique identification is needed to distinguish between two different instantiations of the same bean.

4.2.4 Workspace

The *Workspace* is a local container, which holds all the applications (beans) that are part of a session. When a user joins a session, a workspace for that session is automatically launched and the current state of the session is displayed. When a user closes the *Workspace* windows the user is then logged out of that session.

4.2.5 WorkspaceContents Class

The *WorkspaceContents* class is a very simple class that holds a session name and a list of bean identification numbers. When a new bean is dropped into the workspace the *Session* gives the new bean a BeanID and then adds the BeanID to the *WorkspaceContents* list. This BeanID is used to uniquely identify a bean in the shared space.

4.2.6 UserAccount Class

The *UserAccount* class defines a collaborant within the system. Information such as name, and logon time are all stored in the *UserAccount* objects. This information is needed to maintain control over the collaborative environment and provide awareness information. Users logon when they desire to participate and logoff the system when they are finished. The *UserAccount* class also plays an important role in keeping track of the state of the collaborant in the environment, by maintaining a list of sessions in which they are participating. When the user logs out of the system, the user is automatically dropped from every session he was participating in. In this way, consistency is maintained in the system.

4.3 Event Processing

Any collaborative system relies on the framework or application to pass information back and forth to enable communication between users. This is done primarily through events. The remote event model, discussed in Section 4.3.2, is used because it handles the distributed concerns of synchronization, consistency, unreliability, and unordered processing. JavaSpaces™ implements the remote event model and makes it a good choice for implementation. Prior to discussing the event model and the listeners that support that model, it is important to discuss how synchronization is achieved in the framework.

4.3.1 Synchronization Mechanism

To make the collaboration work between clients, the clients must be synchronized. The design indicates a four-step process of taking or locking the object, manipulating it, writing it to the global space, and then reading the new object on each client. JavaSpaces™ technology provides a built-in mechanism to

perform each of these functions. The following operations map exactly to this synchronization model: *take*, *write*, and *read*.

The *take* operation removes the object from the space, thus, locking it for exclusive use to the client who took it. Once exclusive access to the object is gained, the object can be manipulated. Upon completion of some action by the user, the object is then written back to the space as a new object by the *write* operation. When an object is written to the JavaSpace, registered listeners recognize a change and notify all the clients of the update. This function is performed by the *notify* operation. Upon notification all the participants in a session read the object using the *read* operation.

4.3.2 Distributed/Remote Event Model

To understand how the *notify* method operates it is important to have a firm understanding of events and event models. "Events are the mechanism used for asynchronous communication in Jini™ systems, and as such are crucial to the effective use of the technology" [45]. The distributed event model provides the necessary protocol to allow for clients to communicate with one another, thus enabling a collaborative framework to be built.

To help understand the distributed event model, it is important to first understand how the delegation-based event model works and why it is insufficient for distributed systems. In the delegation-based event model there is no central dispatcher of events; every component that generates events dispatches its own events to registered listeners as they occur. The *Event Listener* first registers with the *Event Source* as a listener for a particular event. Once the event occurs the *Event Source* fires an *Event* object to the *Event Listener* (see Figure 24). The *Event Listener* then processes the *Event* object to produce the desired effect from the event.
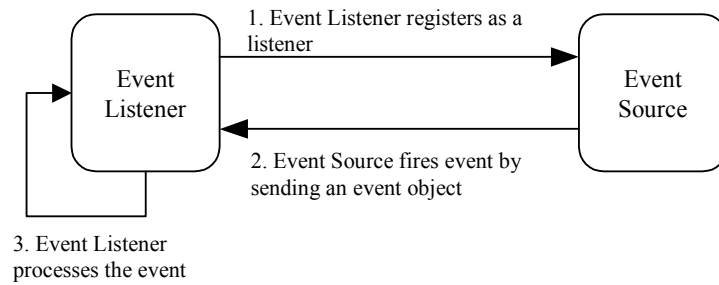
Figure 24. Delegation-base Event Model [56]

Simon Roberts and Jon Byous [45] presented a number of reasons why the existing delegation-based event model is insufficient. These reasons are summarized here:

- First the pre-existing listener interfaces (*ActionListener* and others) do not throw a *RemoteException* from their handler methods. Since Jini™ technology is tied together using Java Remote Invocation (RMI) this exception must be thrown.   But, Java™ prohibits overriding or implementing a method such that the new method throws exception not declared in the original method.

- Second, the *Event* classes, such as *ActionEvent* and others, contain references to the source objects that may be used in processing the event. This reference causes a problem, due to the fact that in some cases the event source object is non-serializable. This prevents the use of the source object as an argument to a remote method call.

- Third, the delegation model assumes that delivery of events is synchronous, reliable, ordered, and quick. This assumption works in a single Java virtual machine, but is not guaranteed in a distributed environment. Messages may be lost, duplicated, reordered or corrupt, thus a new event methodology must be used to overcome these problems.

- Lastly, the registration of a listener wishing to receive an event is handled only by explicit calls, such as *addWindowEventListener* and *removeWindowEventListener*. In a distributed environment the temporary failure of the listener will cause deregistration of the event,

without the explicit call to the *removeWindowEventListener*. Thus, this approach is insufficient for a distributed environment.

Jini™ addresses these difficulties in the distributed event model. First, Jini™ introduces a single "universal" event listener: the *RemoteEventListener* interface. This is a remote interface, indicating that Remote Method Invocation (RMI) is used to call the *notify(RemoteEvent)* method that the interface defines. This method then notifies the listener about an event.

The second and third problems described above are handled by the elegant design of the *RemoteEvent* object. The Jini™ *RemoteEvent* object is defined with only four fields (*source*, *eventID*, *seqNum* and *handback*). The first field is a handle on the source of the event and is generally a remote reference. The *eventID* is an identifier for the event that gives the handler method a convenient mechanism for determining the reason the event has been delivered. The *seqNum* is a sequence number that is used to give some indication if an event is delivered in the correct order. The distributed model then uses this information to ensure that the listeners get all the messages or events in the correct order. The fourth and final field, the *handback*, or registration, object is an optional parameter.

The final problem that was posed with the delegation model is how to abandon event registration when a listener fails. This is handled by leasing. Leasing is a core part of the Jini™ programming model. It provides the mechanism for controlling garbage collection at the API-level. In other words, establishing a lease for an event registration provides a means for a timeout to that registration. If the leaseholder (usually the event listener) keeps renewing the lease at suitable intervals, then the event source will maintain the registration; however, if the lease is allowed to expire or is cancelled explicitly by the leaseholder, then the event source will recognize that a problem has occurred and will de-register the event listener automatically. Thus, this solution overcomes the inability of the delegations to detect failures and abandon event registrations.

The distributed event model uses four steps to register event listeners with event sources. Figure 25 depicts the registration steps and remote event processing. The first step is for the registrant to register

the remote event listener with the event source. The event source or generator then returns an *EventRegistration* object to the registrant that is then sent to the remote event listener. This makes the connections necessary for the event generator or source to send a *RemoteEvent* to the remote event listener. This model is the basis for processing changes made in the collaborative distributed environment.
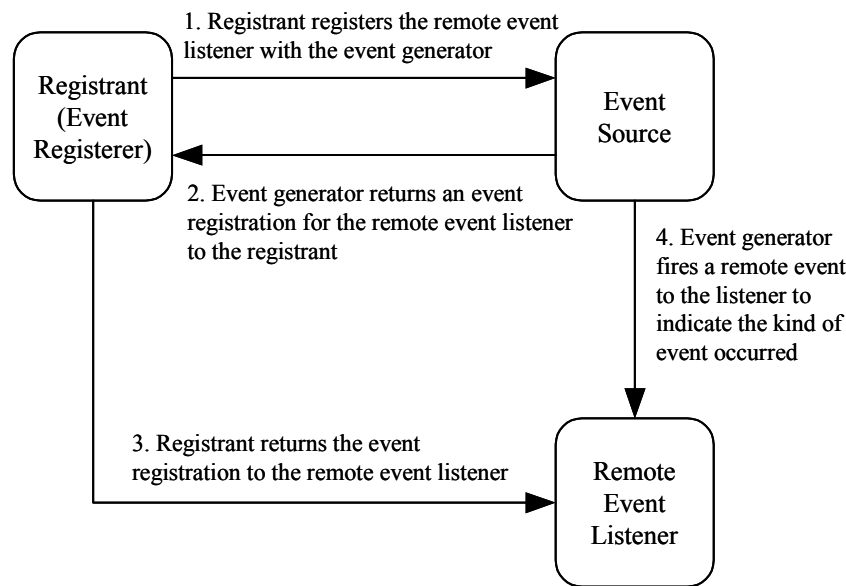
Figure 25. Jini™ Remote Event Model [56]

The distribute event model is the basis of the *notify* operation. That is, that a registrant registers an event listener with an event generator. In this case, the event source is the JavaSpace. The event registrant is the class that registers the remote listener. The remote event listener receives notification when a change is made to an object. The registrant calls the *notify* method on the JavaSpace with two main parameters: the *Entry* and the listener. The *Entry* object is the template that is used to match objects against future writes to the space and the listener is the object that receives the remote event sent by the event source. The JavaSpace then notifies the remote event listener when an object that matches the template is written to the space.

To facilitate collaboration there are three main listeners that are used to handle events within the framework: *SessionManagerListener*, *WorkspaceContentsListener*, and *BeanListener*. Each of these listeners is discussed in greater detail below.

4.3.3 SessionManager Listener

The *SessionManagerListener* is registered to the shared space by the *Desktop* to listen to changes to the *SessionManager* object. Its sole purpose is to process changes that are made to the *SessionManager*. The listener must implement the Jini *RemoteEventListener* interface and extend the RMI *UnicastRemoteObject*. The *RemoteEventListener* interface is used to indicate to the framework that the *notify()* method is implemented. As part of the remote event model, the remote listener *notify()* method is called when an object is written to the shared space that matches the template. Thus, the *notify()* method must be implemented in the listener class or an error will occur. Figure 26 shows that the *notify()* method is the only implemented method of the *SessionManagerListener*. In this instance, the *notify()* method simply reads the new *SessionManager* class from the JavaSpace and notifies the *Desktop* to update its monitor display. The *Monitor*, *MonitorTree*, and *MonitorTreeModel* classes take the information stored in the *SessionManager* and display it as a *JTree*, as shown in Figure 20.
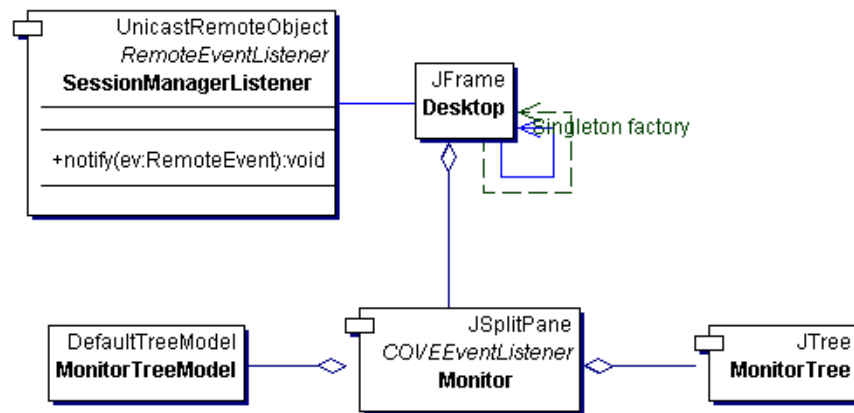
Figure 26. *SessionManagerListener* UML Diagram

The listener must also implement the RMI *UnicastRemoteObject* because the event processing is done via Java's Remote Method Invocation (RMI). Thus, this interface must be implemented to allow for the *notify()* method to be called remotely. The *UnicastRemoteObject* class adds the needed functionality to enable the listener to communicate across a network to other clients. This implementation decision requires that all listeners be compiled using Java's RMI compiler (rmic). The output of this compiler is a

stub and a skeleton bytecode file that must be placed in the root directory of the running Hypertext Transfer Protocol (HTTP) server. The reason for the specific location of the stub and skeleton files is due to the fact that the lookup server looks for these files in that location. If it cannot find them, then the framework will cease to work because the notify methods will never be run, thus never allowing the collaboration mechanism to operate. For further information concerning RMI see Appendix B.

4.3.4 Workspace Contents Listener

The *WorkspaceContentsListener* is registered to the shared space by the *Workspace* class and listens for changes made to the *WorkspaceContents* objects. For each *WorkspaceContents* there is a unique listener associated to it. Thus, the appropriate listener is sent a remote event when changes to the *WorkspaceContents* object are written to the JavaSpace. The notify method must be implemented as discussed above (Figure 27 shows the *notify* method as the only method the listener implements).

This *notify()* method simply calls the *Workspace* to rebuild itself and redisplay its contents. The *Workspace* uses the *WorkspaceContents* list to keep its collection of beans up-to-date. Each bean is contained in a *BeanData* object, which is just a *JInternalFrame* object. The rebuild process iterates through the *WorkspaceContents* list and ensures that the *Workspace* has all beans in its workspace. If it does not, the *Workspace* loads the bean and adds it to the workspace. If a bean is not in the list and the *Workspace* has it instantiated, then the *Workspace* removes the bean from the workspace. Once everything is up-to-date in the workspace, the redisplay method is called. This method simply repaints all the components in the workspace.

It is important to note that the *Workspace* is never written to the shared space, but remains local on each client. The *BeanData* object, however, is decomposed and written to the space. The actual bean that is instantiated is shared through the JavaSpace. The bean is written to the space and then read by clients participating in that session and added to their local workspace. The *BeanListener* class is discussed in Section 4.3.6.
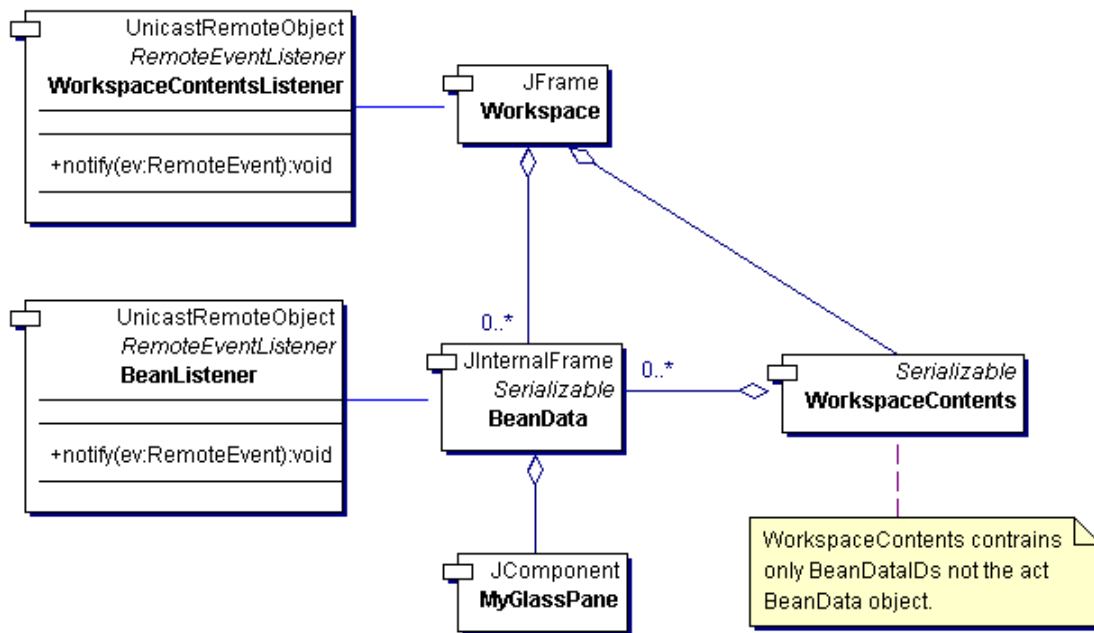
Figure 27. *Workspace* Components UML Diagram

4.3.5 Glass Pane

To be able to capture events, a glass pane is used to intercept all user events (mouse and keyboard events). Once events are captured, the synchronization mechanism in the framework is notified. The Java™ tutorial provides this insight: "The glass pane is useful when you want to be able to catch events or paint over an area that already contains one or more components. For example, you can deactivate mouse events for a multi-component region by having the glass pane intercept the events" [50]. Figure 28 shows how the glass pane is set at the topmost Z-order to cover the bean's graphical user interface and intercept all user events, without occluding the underlying Java™ components.

As each bean is dropped into the *Workspace,* a glass pane is added to the *JInternalFrame* that encapsulates it. This keeps each bean separate, enabling events to be intercepted only when a user is interfacing with the bean.

75
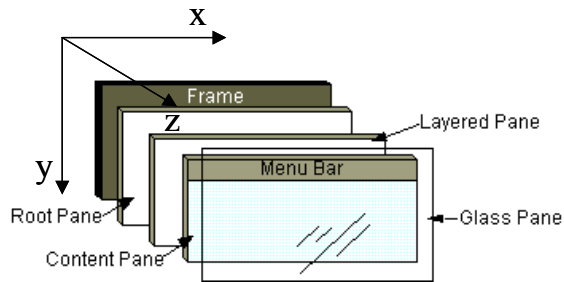
Figure 28. Glass Pane Diagram [50]

4.3.6 Bean Listener

Figure 27 shows a relationship between the *BeanListener* and the *BeanData* classes. When a bean is added to the collaborative environment the bean is encapsulated in a *BeanData* object and stored out in the space via the *BeanEntry* object. The *BeanData* class registers the listener with the shared space to listen for changes to the bean. The user's actions with the beans are captured by the glass pane and then passed to the underlying bean component that should receive the events. The bean processes the events and then the resulting bean is written to the space. Just as before, the *notify()* method of the *BeanListener* is then called to process the changed object.

The *notify()* method of the *BeanListener* reads the new bean from the space and updates the workspace with the new bean. After the update takes place, the redisplay method is called to reflect the change in the workspace.

4.4 Remote Code Access

The capability to expand the framework is an important element of this design. To enable expansion, the framework was designed to decouple the applications from the collaboration elements. Thus, to add capability is as simple as adding a bean to the library.

Adding beans is as simple as placing files in the correct location on the server. The dynamic loading mechanism uses a simple hypertext transfer protocol (http) server that stores the class files needed in the serialization process. When an object is passed in or out of a remote method call or is passed to the

JavaSpace, the object is serialized. Only member data within that object is written to the byte stream, not the actual code that implements the object. Thus, the class file must be located upon deserialization of the object.

"When the sender of an object serializes that object for transmission to another Java Virtual Machine (JVM), it annotates the serialized stream of bytes with information called the codebase" [15]. The codebase is the mechanism for remote class loading to find the new classes. It contains information telling the receiver where the implementation of the object can be found, thus enabling the remote virtual machine to deserialize that object. Further information about codebase can be found in Keith Edwards' book "Core Jini, Second edition" [15].

The framework uses this codebase concept to expand the local classpath. The codebase is set, via a property to the server's Java Virtual Machine, that contains the http Universal Resource Locator (URL) that indicates the location of the codebase, for example: -Djava.rmi.server.codebase= "http://129.92.20.80:4220/". The capability enables the framework to dynamically load bean code from a central location, thus making the solution more robust.

4.5 Transactions – Solutions to Consistency and Partial Failures

The most challenging aspect of a distributed system is the difficulty of detecting and dealing with partial failures. These partial failures create inconsistencies in the distributed system. To help prevent these inconsistencies, the notion of a transaction is used.

Transactions keep the objects in the shared space in a consistent state. Either they get changed or they do not. Nothing in between can happen. So if something crashes when trying to change an object the state of the shared space will remain intact because every operation performed by the *SessionServiceProxy* is done through transactions.

Transactions are implemented in every method of the *SessionServiceProxy*. Each method in the JavaSpaces™ interface has a transaction parameter that can be used. The session service uses this

parameter to group several operations together to maintain consistency in the global space. A simple example will illustrate how this works.

Consider a user trying to log on to the system. The login command is issued from the *Desktop* class to the *SessionServiceProxy* via the concrete command *LoginCmd*. The *SessionServiceProxy* is responsible to ensure the JavaSpace remains in a "safe" state, so it uses a transaction. The code in Figure 29 creates a *Transaction* variable called *txn*. It uses the helper classes of the *TransactionFactory* to create a transaction. The variable *m_mgr* is a handle to the transaction manager and is passed into the *create* method of the *TransactionFactory* so it can manage the transaction

```
1.   try
2.   {
3.     trc = TransactionFactory.create(m_mgr, 300000);
4.   }
5.   catch (Exception e)
6.   {
7.     System.err.println("Could not create transaction " + e);
8.   }
9.   Transaction txn = trc.transaction;
```

Figure 29. Transaction Creation Code Example

The transaction now may be used to group like commands together. In the case of a user logging into a system, we want to make sure the user gets added to the system properly (the code for this is shown in Figure 30). First the *SessionManager* must be taken from the space under a transaction (line 5). After it is removed from the shared space, it can be manipulated. In this case, line 9 adds a user, and line 10 updates the session manager on the local machine. Finally, after all the manipulations are complete, the *SessionManager* is written back to the space under the same transaction to become available to all users, (line 12).

If at any time an error occurs, the catch block (lines 15-18) catches the exception and aborts the transaction. If all goes well, the transaction commits (line 19) and the object is written to the JavaSpace.

```
1.  try
2.  {
3.    try
4.    {
5.      sessionManager = (SessionManagerEntry) m_space.take(sessionManTemp, txn, long.MAX_VALUE);
6.      if (sessionManager != null)
7.      {
8.        // Add the user to the sessionManager
9.        sessionManager.m_sessionManager.addUser(userAccount);
10.       SessionManager.setInstance((SessionManager)sessionManager.m_sessionManager);
11.       // Write the SessionManager back to the space
12.       m_space.write(sessionManager, txn, Lease.FOREVER);
13.     }
14.   } // End try (1st)
15.   catch (Exception e)
16.   {
17.     txn.abort();
18.   }
19.   txn.commit();
20.   } // End try (2nd)
21.   catch (Exception e)
22.   {
23.     System.err.println("Transaction commit failed");
24.     ...
25.   }
```

Figure 30. Login Transaction Code Example

V. Results

A comparison of COVE to other frameworks is performed to evaluate the effectiveness of the developed framework. The criteria described in Section 3.1.1 provide the objective categories on which to compare and contrast the different frameworks. This chapter examines the research results and analyzes them with respect to the established criteria.

5.1 Comparative Analysis

This section evaluates each of the described frameworks (COVE, DISCIPLE, COAST, Habanero, and ColVis) with respect to the established requirements listed in Section 3.1.1. A subjective analysis is performed on each framework to measure how well it meets the criteria. When a framework meets that requirement, it is assessed as a strength or weakness to approximate the degree in which the requirement is met. The purpose of the comparison is to see to what degree the new framework meets the described goals and to compare it to other existing frameworks.

5.1.1 Criteria Satisfaction

Generality is the ability of a framework to be used across many different domains. Each framework under consideration provides a backbone that can support any domain. The solution for generality is implemented differently for each framework. The COVE and DISCIPLE frameworks rely on JavaBeans™ applications to support any domain. Habanero takes any Java application and adds code to it to make it collaborative. The COAST framework provides a generic set of components that are used to construct domain-specific applications. The ColVis framework relies on a set of visualization applications that interface with components in the collaborative framework to facilitate generality.

Automation is the spontaneous support of making an application collaborative with minimal intervention from the developer. COVE enables existing beans to be dropped into a framework without any modification. DISCIPLE does a similar thing, but dropping "unaware beans", beans that do not have

any DISCIPLE component extensions, into a workspace causes the framework to crash. This may be due to the version of DISCIPLE that was used in the evaluation. Habanero has an external automated process that converts Java applications to collaborative applications. All of these frameworks require very little effort on the part of the developer to create multi-user applications and are considered to be strengths of the frameworks. COAST supports automation by providing a robust set of components, but still requires extensive developer interaction to build a collaborative application from the ground up. The ColVis framework provides a set of interfaces and classes that are used to create multi-user applications. This approach still requires extensive code changes in the toolkits, and effort on the part of the developer to create new distributed applications.

Shared state and consistency fit naturally together, because they both affect the ability of the framework to effectively support collaboration. These criteria are strengths for each of the frameworks due to their ability to keep shared resources in the same state. Shared state and consistency are maintained in different ways, but each framework effectively implements the criteria.

Scalability is the ability of the framework to expand to handle the load of additional users. This criterion is considered to be a strength for the COVE, COAST, and ColVis frameworks due to their ability to add additional resources to expand their capability to handle a higher load. COVE allows for additional services and shared repositories to be added, while COAST adds additional mediators to expand their capability. The ColVis framework could be modified to add additional JavaSpaces™ to support a higher load. The other two frameworks provide average support for this criterion. Habanero arbitrators use TCP/IP sockets to connect to each client, which do not scale well due to overhead. DISCIPLE provides only one server that may get overloaded and become a bottleneck when numerous clients are passing events.

Robustness is a measure of how well a system handles system failures. The COVE framework provides the most robust solutions for meeting this criterion through the use of Jini™ services. It takes advantage of the self-healing properties of Jini™ by enabling clients to look for alternate services at runtime when a component fails. It allows for redundant services and multiple central object repositories to

be running simultaneously to provide backup when system failures occur. The COAST and ColVis framework provide average support for this criterion. COAST provides mediators that store persistent objects; thus, when these mediators fail, clients may retry to access these objects when the mediator becomes available. ColVis uses a similar persistent storage method through the use of JavaSpaces™. When a client crashes, the current state of a session can be reacquired through reading the event history from the JavaSpace. Both of these approaches still require user action to reacquire the current state of the collaborative environment. Habanero uses an arbitrator to examine each action to ensure that it is legal, thus protecting the framework from actions that may cause failures. However, it does not provide redundancy or persistent storage for state information, thus creating a limitation in the framework. A weakness in the DISCIPLE framework is the lack of attention given to this criterion.

Communication is the ability to transfer explicit and implicit information. Each framework supports explicit communication through the passing and manipulation of entities in the collaborative environment. However, none of them fully address the issues associated with implicit information.

Dynamic loading is the ability to access a common application repository and download the current version of code to execute. The COVE and DISCIPLE frameworks provide a robust solution to this criterion. COVE uses a bean repository and Remote Method Invocation (RMI) technology to access remote class files, thus sending needed code to clients. The DISCIPLE framework uses a resource server to accomplish a similar task. The resource server disseminates needed code to clients involved in a collaborative session. COAST supports dynamic loading at a very limited level. When applications need data, COAST mediators dynamically load data but do not load actual classes. The Habanero and ColVis framework do not support this capability.

Coupling is the measure of how much the application is joined to the collaborative framework. The goals were to make the framework stand apart from the application and to ease the burden of the developers in developing applications. The COVE and DISCIPLE frameworks do this through the use of adapters that capture events. COVE uses a glass pane that is placed over every bean that is dropped into a workspace to capture events and make the connection between the framework and the applications.

DISCIPLE uses adapters that are generated at runtime for each "unaware bean" that connects the framework to the application. These approaches greatly ease the burden of developers, as they need not worry about the collaborative elements of the framework; rather they need only to create beans that have a desired functionality and drop them into the framework. COAST, Habanero, and ColVis provide components that are extended or used in the application to provide the collaborative elements. This approach is the most common approach taken in collaborative frameworks. It enables developers to create applications from the ground up using the set of collaborative components to link the application to the framework.

Frameworks should support a standard language and not rely on additional support from languages (i.e. specialty compilers, interpreters). All the frameworks evaluated use standard languages. COVE, DISCIPLE, Habanero, and ColVis all use standard supported functions from Java. COAST uses Visual Works Smalltalk, which is a platform-independent environment, to provide standard language support. This is a strength for each framework with the exception of COVE. COVE uses the remote method invocation compiler (rmic) that creates an additional constraint in the framework.

Visual collaboration is the framework's ability to support communication at a higher level than that of the data level. This is considered a strength for the COVE and COAST frameworks due to their support of visual abstractions. COVE uses the visual object to collaborate and COAST supports shared models that represent the views. The other frameworks all collaborate via event passing and do not really support the criterion.

Flexibility enables developers to customize components of the framework easily with minimal side effects. COAST, Habanero and ColVis support some type of modular component approach to enable customization of the framework. COAST is built using a set of modifiable components. The side effects caused by modifying a component are difficult to assess, since the software was not obtained for evaluation. Habanero uses an arbitrator that can be customized by the application developer to have the desired functionality, but other components in the framework are tightly coupled; thus, changes would result in side effects. The ColVis framework provides a toolkit that can be customized and changed to meet

83

the needs of the users, but changes to the components of the framework would cause substantial side effects in the developed tools in the toolkit. The COVE and DISCIPLE frameworks have a limited capability to customize framework components. These frameworks use JavaBeans™ to provide great application flexibility to the developers. Modifying the behavior of the framework, however, introduces side effects due to the interrelationships between the framework packages.

Coordination of action is the "floor control" mechanism used to ensure that collaborants take turns. Each framework meets this criterion, but COAST and Habanero provide more robust solutions by supplying several models that change the behavior of control. COAST provides a set of session objects that implement different collaboration modes, such as loosely- and tightly-coupled sessions. Habanero provides several different arbitrators that use different collaboration modes (e.g., turn-taking and student-teacher). This allows the developer the ability to customize the behavior of the coordination of action to ensure the needs of the users are met. The other frameworks take a first-come/first-serve approach by sequentially ordering all actions in the framework. A common server processes requests sequentially as the clients initiate them. These frameworks are effective in providing "floor control" mechanisms, but they lack the ability to facilitate other modes.

Monitoring is the ability of the framework to collect and display information about the collaborants' actions. A strength of the DISCIPLE and Habanero frameworks is their extensive set of tools used to display users' actions and provide user awareness to collaborants. They both use a session manager view to display global information about the environment, but also include tools such as shared pointers and radar views to provide additional insight to what users are doing. The COVE and COAST frameworks primarily provide global session management views to enable users to see what is going on. This is effective but limited because additional context is lost about where other people are working. The ColVis framework provides a session view and a message window, but lacks a global view. This creates a problem when a user wants to find out what sessions are available to join.

Protection provides safeguards on the work of users so information is not inadvertently or purposefully destroyed. This characteristic may seem contrary to the nature of collaboration, but is

important to enable individual work to be accomplished. The DISCIPLE, COAST and Habanero frameworks support this concept to a certain extent. DISCIPLE has the concept of private workspaces that are restricted to their owners. This allows users to work in their own space for a time and then make the session collaborative. COAST and Habanero provide floor control modes that can restrict the access to work within a session. For example, in a student-teacher mode, the teacher is the only one able to initiate actions. This prohibits students from changing the material being presented. The COVE and ColVis frameworks provide limited support to this concept. COVE provides some basic security mechanisms to prohibit unauthorized clients from getting access to applications in the framework. ColVis provides unshared workspaces, but they cannot become shared. Additionally, ColVis provides a moderator that can mute a participant so they are unable to contribute to the collaboration. This criterion is not a strength of any of the frameworks evaluated.

5.1.2 Summary of Analysis

Table 3 summarizes the analysis and identifies the strengths and weaknesses given to the frameworks. The blank spaces in the table indicate that the particular criterion was met to some extent, but was not considered to be a strength nor a weakness of the framework. This table clearly shows that none of the existing frameworks have strengths in every criterion used for evaluation. The reason for this is that each framework centers on a certain set of characteristics and other areas are left underdeveloped. Protection and flexibility are some of those underdeveloped areas for all frameworks.

There is no conclusive evidence that COVE or any of the listed frameworks is better than another. Each framework maintains a certain focus on a particular set of capabilities. For example, COVE and DISCIPLE focus on dynamic loading as a key element to their design, while COAST and Habanero provide better solutions to coordination of action criteria. Therefore, to determine which framework is desired in building a collaborative application, it is imperative to examine the functionality and characteristics desired, and then choose appropriately.

	COVE	DISCIPLE	COAST	Habanero	ColVis
Generic	S	S	S	S	S
Automation	S	S		S	W
Shared State	S	S	S	S	S
Consistency	S	S	S	S	S
Scalability	S		S		S
Robustness	S	W		W	
Communication					
Dynamic Loading	S	S	W	W	W
Coupling	S	S			
Standard Language Support		S	S	S	S
Visual Collaboration	S		S		
Flexibility	W	W			
Coordination of Action			S	S	
Monitoring		S		S	W
Protection	W		W	W	W

Table 3. Frameworks Comparison Analysis Table

5.2 COVE Framework Benefits

The COVE framework uses a robust service mechanism to provide the collaborative functionality of session management. Jini™ technology provides this robustness with mechanisms to publish services that are scalable, reliable, and simple.

The framework is implemented in the Java programming language which provides great flexibility and platform independence. Due to the fact that all Java code is compiled into bytecode, then run on a Java Virtual Machine, all software written in Java becomes independent of the operating system and hardware architecture on which it runs. This provides programming flexibility to run any program or design a framework that will work on any platform. Thus, the framework's compiled code can be run on many different platforms that have JVM installed.

The shared object approach reduces the number of objects that are maintained by storing them in a central location in the collaborative environment. Complex synchronization and concurrency controls are avoided to help maintain consistency throughout the collaborative environment. JavaSpaces™ is a

technology that makes this all possible. In essence it hides the concurrency control and synchronization from the framework by implementing them in the JavaSpaces™ API.

The network and collaboration aspects of the framework are decoupled from the application. Any bean can be loaded and used in the framework. This capability enables developers to focus their attention on developing applications and not worry about the collaborative concerns.

Visualization techniques are supported through the use of shared objects, thus any technique can be employed within this framework. The only constraint to the visualization techniques used is the limitation the application imposes. Thus, techniques such as zooming, and filtering, can be used effectively within the framework.

5.3 Serialization Problem with JavaBeans™

During the implementation an assumption was made with regards to the serializable nature of JavaBeans™. Swing components are built upon the JavaBeans™ architecture. Therefore, they are beans and have the characteristics of JavaBeans™. So the assumption was made that any bean could be serialized to the JavaSpaces™, and then reconstituted in the same state.

The assumption that any bean is by definition serializable was tested and failed. The reason for the failure was a lack of understanding of the serializable mechanisms of JavaBeans™. The JavaBeans™ API documentation states, "as part of JavaBeans™ 1.0 we support the Java object serialization mechanism which provides an automatic way of storing out and restoring the internal state of a collection of Java objects" [52]. This mechanism is used to facilitate the goal of persistent storage of a bean. "However a bean should not normally store away pointers to external beans (either peers or a parent container) but should rather expect these connections to be rebuilt by higher-level software" [52]. This last statement creates the problem and requires a work around to have the beans share state in the JavaSpace.

To solve the problem, a constraint on the construction of bean applications is imposed. The constraint levied on each bean is to implement the *readObject* and *writeObject* methods. The *writeObject*

method is responsible for writing the state of the object for its particular class so that the corresponding *readObject* method can restore it. The *readObject* method is most critical, because it restores the connections to other external beans (i.e. reestablishing listeners). The trick to solving the problem is that the Java Virtual machine automatically checks to see if either method is declared and makes the appropriate call. This ensures that the integrity of the class is maintained and the serialization protocol can continue. This solution is effective but reduces the overall flexibility and generality of the framework.

VI. Conclusions and Future Work

The previous chapters describe the design and implementation of the Collaborative Visualization Environment (COVE) framework, and provide a comparative analysis to existing frameworks. This chapter summarizes the results from the previous chapters and suggests some potential areas for future work to enhance the COVE framework.

6.1 Conclusion

The goals of this research were to create a generic framework that supports shared interaction, visual sharing, remote code access, and easy tool integration.

- Shared interaction – the collaboration between geographically separated users interacting with data and visual representations to accomplish tasks.

- Visual sharing – remote users collaborate at a higher level of abstraction than the data through the sharing of visual objects.

- Remote code access –users access remote data and applications without the need for previous installation.

- Easy tool integration – flexibility to easily integrate any Java tool and make it collaborative.

- Facilitate software development – the purpose of a framework is to ease the burden of developers in developing software

These goals were met through the use of a centralized object repository (JavaSpaces™), Jini™ services, and JavaBeans™ technology. By meeting these goals, the collaborative visualization environment (COVE) framework provides a simple and effective means for rapid development of collaborative visualization applications.

The COVE framework provides several advantages to developers of collaborative applications. It is a generic framework for synchronous collaboration of users with heterogeneous computing platforms. The framework enables shared interaction between geographically separated users through the interaction of visual objects. Object sharing is made possible through the integration of JavaSpaces™ into the framework. It facilitates the storage and retrieval of any type of serializable object with relative ease and efficiency. JavaSpaces™ also maintains consistency and concurrency through the use of this common object store. All collaborants access this shared repository to gain access to the current state of a collaborative session.

Remote code access enables applications to be run on remote machines without the need of previous installations. This capability is achieved through the use of Jini™ technology and Remote Method Invocation (RMI). Jini™ provides a mechanism for services to be available for use from anywhere on the network through a lookup server. This enables clients to access service interface code remotely. RMI is another key enabler for remote code access through its extension of the classpath, called the codebase. This extension enables clients to download remote code from a specified location to use locally on their machines. This capability is used in the bean repository by placing all bean class files in the codebase location to enable any client access to the code. These capabilities add great flexibility and support to the framework by enabling clients to gain access to needed software through dynamic loading.

Another advantage of this framework is the simplicity with which beans can be integrated and made collaborative. The framework is designed to completely decouple collaborative elements from the applications. This separation allows developers to concentrate their efforts on building JavaBeans™ applications. To integrate the new application into the framework, the developer need only place the new bean in the bean repository and then load it in the collaborative environment.

Due to problems with beans serialization, the achieved flexibility of the framework was less than desirable. It was assumed that any bean could be serialized; however this turned out to not be true in all cases. To get around this problem, the *readObject* and *writeObject* methods had to be used to reconnect objects (e.g., listeners, etc.), thus forcing the bean designer to consider serialization during development.

Sun Microsystems, Inc. recently released a new version of Swing that changes the persistence mechanism for JavaBeans™ from general binary serialization to a schema using XML. This approach is intended to establish a standard for design and address the long-term persistence problems for JavaBeans™, thus allowing developers to overcome the serialization limitations.

An established set of criteria was used to measure how well the goals were met. The criteria were based on fifteen characteristics, derived from previous work, that are desired in a collaborative visualization framework. The COVE framework had nine strengths and two weaknesses and satisfied all the goals to some degree. Of the frameworks compared, COVE provided the most strengths and is the most suitable framework to use for the established criteria.

6.2 Future Work

There are two main areas where future work can be completed to improve the framework. The first is to enhance the framework to support additional collaborative capabilities (i.e. component customization, protection consideration, etc.). The second is to research new ways to add visualization tools in the framework to provide users with greater awareness within the collaborative environment.

6.2.1 Enhancements

6.2.1.1 Customizable Components

The design of the system dictates certain behaviors of the system, such as allowing every member of a session equal access. This situation may be unsuitable if tighter controls are needed. For example, in the teacher-student mode, the teacher must have complete control of all actions; otherwise students may hinder the learning by changing the state of the environment prematurely. Thus, to increase the flexibility of the framework, this type of functionality should be consolidated into a single customizable component. This also applies for other elements of the framework that may need to be customized by the application developer. This allows developers the ability change the behavior of a component to meet the needs of their organizations without affecting the other elements of the framework.

6.2.1.2 Protection Mechanisms

One obvious potential problem with a shared workspace is the inadvertent altering of work by another in the workspace. The framework should support mechanisms for locking different elements or regions of the shared workspace for private use. This would enable users that are solving a problem a reserved space where they can manipulate objects prior to sharing them with all members of the session.

6.2.1.3 Session Recording and Playback

Another feature that could enhance the session management is the ability to keep a record of all user actions and generate a session history. This capability could be used to produce playbacks of a collaborative session to provide information on how a decision or consensus was reached.

6.2.1.4 Complex Bean Integration

To evaluate the performance of the framework, the desired approach was to use a complex bean, such as mission planning or whiteboard applications. Due to the time constraints, only simple beans were used to demonstrate the functionality of the framework. With this limitation identified, a valuable future effort would be to integrate more complex beans to evaluate the framework's ability to scale to more complex applications. Along with the evaluation, a bandwidth analysis should be performed to determine the amount of data being passed over the network.

6.2.1.5 Passing of Object Differences

Based on the design of the framework, every time a bean is changed, it must be serialized over the network. With complex beans, this could get very time and bandwidth intensive. A future effort could examine the feasibility of passing object differences from the shared object space to the clients. This would reduce the potential bottleneck of having large amounts of data being sent over the network. One possible solution is the use of XML. The new persistence mechanism for JavaBeans™ in Java Swing version 1.4 uses XML to store the current state of the beans. This new mechanism could be used to send changes, thus updating the beans based on the differences in the XML data. This solution would greatly reduce the

amount of data transferred across the network by limiting network traffic to changes to the bean instead of the entire bean.

6.2.2 Addition Visual Awareness Capabilities

Awareness facilitates multi-user coordination by providing hints to what and where people are working in a shared environment. Several visualization techniques, such as TelePointers and Radar views [28], have been shown to improve users cohesiveness and effectiveness.

6.2.2.1 TelePointers

TelePointers provide a mechanism for each user to track the mouse pointer of other users within a collaborative session. Each user can assign a label and a color to their pointer. Mouse movement from each participant can be viewed in a shared application window and viewed by all participants in a session.

6.2.2.2 Radar Views

The concept of overview and detail [4] is very useful in providing global awareness to participants in a collaborative session. Radar views, described by Gutwin, et al [21,24], use overview and detail to display information about collaborants interaction on a single screen (Figure 31). The overview shows the entire workspace in miniature, and the objects as they move and change within that workspace. Each collaborants' telepointer and the extents of their main views are added to provide additional detail. Radar views make collaborants' location, presence, and actions visible, regardless of where they are located in the workspace.
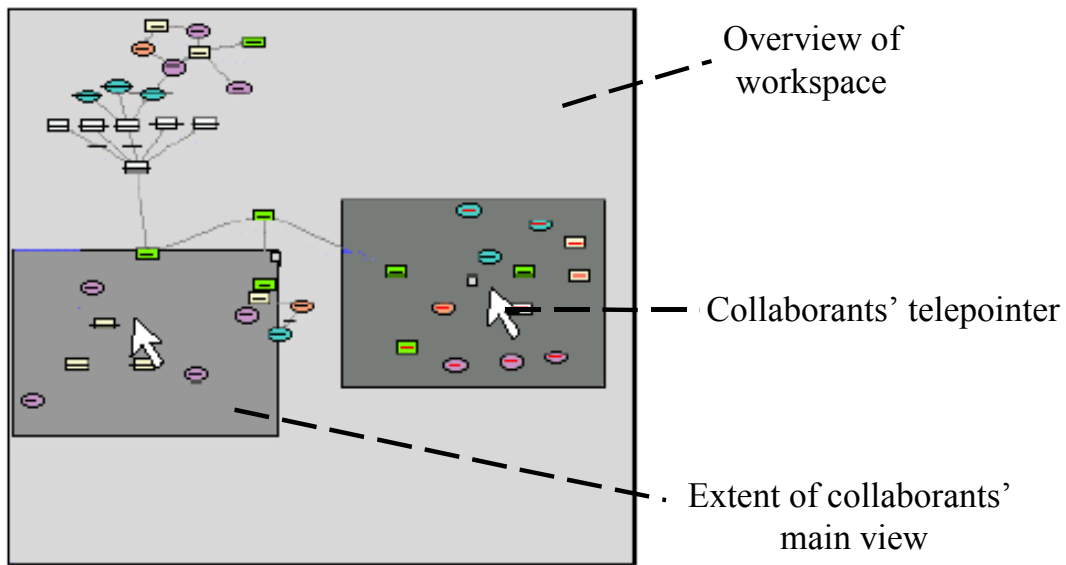
Figure 31. Radar view example [21]

6.3 Final Concluding Thoughts

This collaborative visualization framework provides a small piece for a Joint Battlespace InfoSphere (JBI) solution to enable the sharing of information. This capability can aid the warfighter in making decisions and fulfill the mission of Joint Vision 2020, which is to fight and win wars.

The COVE framework is a simple and effective means for rapidly developing collaborative visualization applications. The use of Jini™ and JavaSpaces™ provides system robustness and effectively enables the sharing of Java objects. The framework has met all established goals and should be considered a success.

A. Appendix A - UML Diagrams

Appendix A contains the UML diagrams, showing the relationships between classes used in the framework.

A.1 Relevant System Packages

cove.workspace
AwareGlassPane
+AwareGlassPane PointerAA
+BeanData
+BeanListener
+BeanListener Skel
+BeanListener Stub
+*FreePane*
+*IFreePane*
+*IWorkspaceContents*
+MyGlassPane
+PointerEvent
+*PointerListener*
+RadarView
+SessionListener
+SessionListener Skel
+SessionListener Stub
+*Singleton*
+Workspace
+WorkspaceContents
+WorkspaceContentsListener
+WorkspaceContentsListener Skel
+WorkspaceContentsListener Stub

cove.desktop
+BaseInfoPanel
+BrowserControl
+Desktop
+FocusOnAdapter
+*IFocusOnListener*
+*ITreeCell*
+Monitor
+MonitorTree
+MonitorTreeModel
+RootCell
+SessionCell
+SessionManagerListener
+SessionManagerListener Skel
+SessionManagerListener Skel
+SessionManagerListener Stub
+SessionManagerListener Stub
+Standard
+TextDialog
+UserCell

cove.jspace
+BeanEntry
+BeanEntry1
+Channel
+ChannelEntry
+ChannelMessageEntry
+*IChannel*
+*ISession*
+*ISessionManager*
+*ISessionService*
+Index
+Message
+Session
+SessionEntry
+SessionManager
+SessionManagerEntry
+SessionServiceProxy
+SessionServiceWrapper
+SharedVar
+SpaceAccessor
+TransactionManagerAccessor
+UnknownUserException
+UserAccount
+UserAccountEntry
+UserExistsException
+WorkspaceContentsEntry

cove.loader
+*ByteCodeTransformer*
+*IJarInfo*
+*IJarLoader*
+*InputStreamProducer*
+JarInfoImpl
+JarLoaderImpl
+JarStreamLoader
+ObjectInputStreamLoader
+TheClassLoader

cove.toolbar
+BeanInfoIndexCell
+BeanRootCell
+BeanToolbar
+BeanTree
+BeanTreeModel
+*IBeanTreeCell*
+JarChooser
+JarDialog
+JarInfoCell

cove.commands
+*Command*
+CreateSessionCmd
+JoinSessionCmd
+LeaveSessionCmd
+LoginCmd
+LogoutCmd
+RemoveSessionCmd
+ShutdownCmd

Figure 32. COVE System Packages

95

A.2 commands Package

Figure 33. Commands Package Classes

A.3 desktop Package

JSplitPane
COVEEventListener
Monitor

-m_desktop:Desktop
-m_mtmodel:MonitorTreeModel
-m_tree:MonitorTree

+Monitor
+buildMonitorTree:void
-moveUsersOut:void
+sessionChanged:void
+sessionCrashed:void
+sessionCreated:void
+sessionCreated:void
+sessionDropped:void
+sessionJoined:void
+userEnterSession:void
+userEnterSystem:void
+userLeaveSession:void
+userLeaveSystem:void
+usersCrashInSession:void
+usersLeaveSession:void

#TreeSelectionHandler

selected:String

DefaultTreeModel
MonitorTreeModel

-DEBUG:boolean

+MonitorTreeModel
-findChild:ITreeCell
+findSession:SessionCell
+findUser:UserCell
+insertSession:void
-insertTreeCell:void
+insertUser:void
+insertUser:void
+removeAllCells:void
+removeSession:void
-removeTreeCell:void
+removeUser:void
+removeUser:void

TreeModel uses the ITreeCell classes as the nodes in the model

MutableTreeNode
interface
ITreeCell

+*toString:String*

chatView:Component
infoView:Component

JTree
MonitorTree

-DEBUG:boolean

+MonitorTree

#TreeHandler
#XTreeCellRenderer

selectedItem:String

DefaultMutableTreeNode
SessionCell

-m_session:Session
-m_ssi:ISessionService
-m_view:Component

+SessionCell
+SessionCell
-buildView:Component
+toString:String

chatView:Component
infoView:Component
session:Session

DefaultMutableTreeNode
RootCell

-m_name:String
-m_view:Component

+RootCell
-buildView:Component
+toString:String

chatView:Component
infoView:Component

DefaultMutableTreeNode
UserCell

-m_userAccount:UserAccount
-m_view:Component

+UserCell
-buildView:Component
+toString:String

chatView:Component
infoView:Component
userAccount:UserAccount

JFrame
Desktop

-m_btb:BeanToolbar
-m_currentUser:String
#m_discovery:LookupDiscovery
-m_fols:Vector
-m_hh:HelpHandler
-m_instance:Desktop
#m_monitor:Monitor
#m_serviceTemplate:ServiceTempl
#m_sessionManagerListener:Sess
#m_sessionServiceInterface:ISess
#m_sm:SessionManager
-m_space:JavaSpace
-m_userColor:Color
-m_wsTable:Map

UnicastRemoteObject
RemoteEventListener
SessionManagerListener

-m_desktop:Desktop
-m_space:JavaSpace

+notify:void

JPanel
BaseInfoPanel

-DEBUG:boolean

+BaseInfoPanel

#IconPanel

The Monitor is JSplitPane. The right side contains the Monitor Tree, which provides a view of the session and users. The left component contain information (BaseInfoPanel) about the selected item in the MonitorTree.

Singleton factory

Figure 34. Desktop Package Classes

Standard

+Button(iconName:String,text:String
+Button(text:String,actionCmd:Strin
+CheckBoxMenuItem(text:String,act
+ComboBox(tip:String):JComboBox
+Font():Font
+GrayedImageIcon(image:Image):I
+ImageIcon(image:Image):ImageIc
+ImageIcon(iconName:String,width
+ImageIcon(iconName:String):Imag
+Label(text:String,alignment:int):JL
+Label(text:String):JLabel
+List(data:Vector,tip:String):JList
+List(tip:String):JList
+Menu(text:String,actionCmd:String
+MenuBar(tip:String):JMenuBar
+MenuItem(text:String,actionCmd:S
+Panel():JPanel
+PasswordField(columns:int,tip:Str
+RadioButtonMenuItem(text:String,
-Standard()
+TextArea(rows:int,columns:int,isE
+TextField(columns:int):JTextField
+TextField(columns:int,isEditable:b
+ToggleButton(iconName:String,tex
+ToolBar():JToolBar
+moveCursor():Cursor
+normalCursor():Cursor
+resizeEWCursor():Cursor
+resizeNECursor():Cursor
+resizeNSCursor():Cursor
+resizeNWCursor():Cursor
+resizeSECursor():Cursor
+resizeSWCursor():Cursor
#setLookAndFeel():void
+setLookAndFeel(component:Com

This class provides a one shop stop for GUI components that are used in the Framework. Thus, maintaining consistency in the GUI

JDialog
TextDialog

-m_columns:int=20
-m_input:String=""
-m_okButton:JButton
-m_owner:Component
-m_selectedButton:String="Canc
-m_tfInput:JTextField

+TextDialog(owner:Component,
+TextDialog(owner:Component,

#CancelHandler
#DocumentHandler
#OKHandler

input:String
selectedButton:String
visible:boolean

Alternative extention to the JDialog box, that places text in the textfield. Could be used instead of the JOptionPane.

java.rmi.server.RemoteStub
net.jini.core.event.RemoteEventListe
java.rmi.Remote
SessionManagerListener Stub

-interfaceHash:long
-operations:java.rmi.server.Operati
-serialVersionUID:long
-useNewInvoke:boolean

+SessionManagerListener_Stub(ja
+SessionManagerListener_Stub()
+notify(:net.jini.core.event.RemoteE

java.lang.Object
java.rmi.server.Skeleton
SessionManagerListener Skel

-interfaceHash:long

+SessionManagerListener_Ske
+dispatch(:java.rmi.Remote,:java

operations:java.rmi.server.Oper

RMI Stubs and Skeltons for the Remote Implmentation object.

Figure 35. Utility Classes used in cove.desktop Package

A.4 loader Package

JarStreamLoader

+DEBUG:boolean=false
+WARNEDABOUTNOBEANS:bo
-m_Beans:Hashtable=new Has
-m_ClassList:Vector=new Vecto
-m_JarInfo:IJarInfo=null
-m_JarName:String
-m_Loader:TheClassLoader
-m_SerList:Vector=new Vector()

+JarStreamLoader(aJarName:S
+JarStreamLoader(aJarName:S
#createJarInfo():boolean
#loadJarFromStream(anIS:Input
#parseAttributes(aName:String,
#parseEntryData(aName:String,

ObjectInputStream
ObjectInputStreamLoader

+DEBUG:boolean=false
-m_Loader:ClassLoader

+ObjectInputStreamLoader(anIs
-primitiveType(aType:char):Class
#resolveClass(aClassDesc:Obj

---

interface
InputStreamProducer

---

interface
IJarInfo

+createBean(aName:String):Obj
+getBeanInfo(anIndex:int):Bean
+getBeanInfoByName(aName:S
+getBeanName(anIndex:int):Str
+getIndexByName(aName:Strin

Serializable
JarInfoImpl

+DEBUG:boolean=false
-m_BeanInfos:BeanInfo[]
-m_BeanNames:String[]
-m_BeanToJar:Hashtable=new
-m_FromPrototype:boolean[]
-m_JarName:String
-m_Loader:TheClassLoader

+JarInfoImpl(aJarName:String,a
+JarInfoImpl()
+createBean(aName:String):Obj
diagnoseInstantiationException(
+getBeanInfo(anIndex:int):Bean
+getBeanInfoByName(aName:S
+getBeanName(anIndex:int):Stri
+getIndexByName(aName:String
+getJarName(aBeanName:Strin
+isFromPrototype(aName:String

ClassLoader
Serializable
TheClassLoader

+DEBUG:boolean=false
-KEEPLOADING:boolean=true
-LOADERS:Hashtable=new Hashta
+THERESOURCEPROTOCOL:Stri
-m_MimeHash:Hashtable=new Ha
-m_RawClasses:Hashtable=new
-m_ResourceHash:Hashtable=new
-m_StreamOverrides:Hashtable=n
-m_TheName:String
-m_Transformer:ByteCodeTransfo

+CreateLoader(aTheName:String):
+GetLoader(aTheName:String):The
-TheClassLoader(aTheName:Strin
-applyDefinition(aName:String,aRe
+applyDefinitions(aClassList:Vecto
+defineClassFromBytes(aName:St
-getByteArrayFromFile(aFileName:S
+getLocalImageResource(aName:
+getLocalResource(aName:String)
+getLocalResourceAsStream(aNa
+getLocalResourceType(aName:S
+getResource(aName:String):URL
+getResourceAsStream(aName:St
+instantiate(aName:String,anIsp:In
+loadClass(aName:String):Class
#loadClass(aName:String,aResolv
+loadClassFromFile(aFileName:St
#putResource(aResName:String,a
+setByteCodeTransformer(aTransf
+setStreamOverride(aName:String

---

interface
IJarLoader

+loadJar(aJarName:String,aSource
+loadJar(aSource:URL):IJarInfo

JarLoaderImpl

+DEBUG:boolean=false
-m_LoadedJars:Hashtable=new H
-m_LoaderName:String="COVE"

+JarLoaderImpl(aLoaderName:Stri
+JarLoaderImpl()
#doWork(aJarName:String,aSource
+loadJar(aSource:URL):IJarInfo
+loadJar(aJarName:String,aSource

---

interface
ByteCodeTransformer

+select(name:String):boolean
+transform(aByteCode:byte[]):byte

Figure 36. Loader Classes

A.5 jspace Package

interface
ISessionService
+*addBean:void*
+*createSession:void*
+*createUser:UserAccount*
+*distributeBean:void*
+*getBean:BeanEntry*
+*getBean:BeanEntry*
+*getSession:Session*
+*getUserAccount:UserAccount*
+*getWorkspaceContents:WorkspaceC*
+*joinSession:void*
+*leaveSession:void*
+*login:void*
+*logout:void*
+*removeBean:void*
+*removeSession:void*
sessionManager:SessionManager
workspaceContents:WorkspaceCont

Serializable
SessionServiceProxy
-m_mgr:TransactionManager
-m_space:JavaSpace
+SessionServiceProxy
+addBean:void
+createSession:void
+createUser:UserAccount
+distributeBean:void
+getBean:BeanEntry
+getBean:BeanEntry
+getSession:Session
+getUserAccount:UserAccount
+getWorkspaceContents:Works
+joinSession:void
+leaveSession:void
+login:void
+logout:void
+removeAllSessions:void
+removeBean:void
+removeBeans:void
+removeSession:void
+sessionExists:boolean
message:String
sessionManager:SessionMana
space:JavaSpace
transactionManager:Transactio
workspaceContents:Workspace

Runnable
SessionServiceWrapper
#m_join:JoinManager
#m_mgr:TransactionManager
#m_proxy:Object
#m_serFile:File
#m_sessionManager:SessionMan
#m_space:JavaSpace
+SessionServiceWrapper
-clean:void
#createProxy:ISessionService
-init:void
+main:void
#readState:PersistentData
#register:void
#reregister:void
+run:void
+usage:void
#writeState:void
+IDListener
+PersistentData

Figure 37. Session Service Classes

SessionManagerEntry *(Entry)*
+m_sessionManager:ISessionMan

+SessionManagerEntry(smi:ISessi
+SessionManagerEntry()

ISessionManager *(interface, Remote)*

+addSession(session:ISession)
+addUser(user:UserAccount):vo
+getInstance(name:String):ISess
+getUser(name:String):UserAcc
+removeSession(session:ISess
+removeSession(name:String):v
+removeUser(name:String):void
+removeUser(user:UserAccoun

SessionManager *(Serializable)*
-m_instance:SessionManager=nul
-m_mgr:TransactionManager
-m_sessions:Map
-m_space:JavaSpace
-m_users:Map

#SessionManager()
+addSession(session:ISession):vc
+addUser(user:UserAccount):void
+getInstance():SessionManager
+getSession(name:String):ISessio
+getSessions():Collection
+getUser(name:String):UserAccou
+getUsers():Collection
+removeSession(sessionName:St
+removeSession(session:ISessio
+removeUser(userName:String):vo
+removeUser(user:UserAccount):v
+setInstance(sm:SessionManager

Singleton factory

Session *(Serializable)*
+DEBUG:boolean=true
#m_BeanIDSeed:int=0
#m_EventDistributer:IEventDistr
#m_ListenerList:EventListenerL
#m_UnawareSupport:IUnaware
-m_beanEventListenerList:Vecto
#m_channel:Channel
-m_sessionEventListenerList:Ve
#m_sessionName:String
-m_sessionServiceInterface:ISe
#m_users:Map

+Session(sessionName:String,
+Session(sessionName:String)
+addBean(aBeanName:String,a
+addUserAccount(userAccount:
+checkUserNameConflict(userN
#distributeBean(sessionName:
+finalize():void
#fireBeanEvent(aSenderName:S
#fireSessionEvent(aSenderNam
+getUserAccount(userName:Str
+join(user:UserAccount):boolea
+leave(user:UserAccount):boole
+leave(userName:String):boole
#processPosterMessage(aSend
+removeBean(aBeanID:int):void
+removeUserAccount(userNam
+resizeBean(aBeanID:int,aDim:
+start():void
+stop():void
+toString():String

eventDistributer:IEventDistribute
sessionName:String
unawareSupport:IUnawareSupp
userCount:int
users:Collection
workspaceContents:Workspace

beanEvent
sessionEvent

ISession *(interface)*

+addBean(aBeanName:String,a
+addBeanEventListener(aL:Bea
+addSessionEventListener(aL:S
+addUserAccount(userAccount:
+getUserAccount(name:String):
+removeBean(aBeanID:int):void
+removeBeanEventListener(aL:
+removeSessionEventListener(a
+removeUserAccount(userNam
+start():void
+stop():void

eventDistributer:IEventDistribute
sessionName:String
unawareSupport:IUnawareSupp
users:Collection
workspaceContents:Workspace

SessionEntry *(Entry)*
+m_session:ISession
+m_sessionName:String

+SessionEntry(name:String,sessio
+SessionEntry(name:String)
+SessionEntry()

SessionManager only contain names of UserAccounts and Sessions not actual UserAccount and Session object

UserAccountEntry *(Entry)*
+m_name:String
+m_userAccount:UserAccount

+UserAccountEntry(username:Strir
+UserAccountEntry(username:Strir
+UserAccountEntry()

UserAccount *(Serializable, Cloneable)*
+DEBUG:boolean=true
-m_ImageURL:URL=null
-m_NetAddr:String=null
-m_TimeStamp:Date=null
-m_UserName:String=null
-m_mgr:TransactionManager
-m_sessions:Map
-m_space:JavaSpace

+UserAccount()
+UserAccount(aUserName:Strir
+UserAccount(aUserName:Strir
+UserAccount(aUserName:Strir
+clone():Object
+joinSession(session:ISession
+leaveSession(sessionName:S
+toString():String
+validate():void

imageURL:URL
mgr:TransactionManager
netAddr:String
sessions:Map
space:JavaSpace
timeStamp:Date
userName:String

Session only contain names of UserAccount objects not actual UserAccount object

Figure 38. Session Management Classes

101

Figure 39. Various Classes used in cove.jspace Package

A.6 toolbar Package

JDialog
BeanToolbar

-DEBUG:boolean=true
-m_btmodel:BeanTreeModel=ne
-m_desktop:Desktop
-m_index:int=-1
-m_jarInfo:IJarInfo=null
-m_jarInfoTable:Hashtable=new
-m_jarLoader:IJarLoader=new J
-m_size:Dimension=new Dimer
-m_wsNames:JComboBox

+BeanToolbar(d:Desktop)
+loadJar(url:URL):IJarInfo
-loadLocalJars():void
+setVisible(wsName:String):void
+setVisible(b:boolean):void
-sort(s:String[]):String[]

-DropHandler
-LoadJarHandler
-TreeSelectionHandler

jarInfoTable:Hashtable
jarInfos:Enumeration
workspaceNames:Set

DefaultTreeModel
BeanTreeModel

-DEBUG:boolean=true

+BeanTreeModel()
+findBean(jarURL:URL,beanInfoInd
-findChild(parent:MutableTreeNode
+findJarInfo(url:URL):JarInfoCell
+insertJarInfo(cell:JarInfoCell):void

MutableTreeNode
interface
IBeanTreeCell

+toString():String

DefaultMutableTreeNode
BeanRootCell

-m_name:String="available jars an

+BeanRootCell()
+toString():String

DefaultMutableTreeNode
BeanInfoIndexCell

-m_className:String
-m_imageIcon:ImageIcon
-m_index:int
-m_jarInfo:IJarInfo

+BeanInfoIndexCell(jarInfo:IJarIn
+getBeanIcon(bi:BeanInfo):Imag
+toString():String

beanInfoIndex:int
imageIcon:ImageIcon
jarInfo:IJarInfo

JDialog
JarDialog

-m_columns:int=20
-m_input:String=""
-m_okButton:JButton
-m_selectedButton:String="Can
-m_tfInput:JTextField

+JarDialog(owner:Dialog)

#CancelHandler
#ChooseHandler
#DocumentHandler
#OKHandler

input:String
selectedButton:String
visible:boolean

DefaultMutableTreeNode
JarInfoCell

-m_jarInfo:IJarInfo
-m_url:URL

+JarInfoCell(url:URL,jarInfo:IJarI
+toString():String

URL:URL
jarInfo:IJarInfo

JTree
BeanTree

-DEBUG:boolean=true

+BeanTree(btm:BeanTreeModel)

#BeanTreeCellRenderer
#TreeHandler

JFileChooser
JarChooser

-DEBUG:boolean=true
-m_jarExtension:String="jar"
-m_jarIcon:ImageIcon=Standard.Im

+JarChooser(directory:String)
+JarChooser()
-getExtension(file:File):String
+getSelectedJar(parent:Componen
-isJar(file:File):boolean

JarFilter
JarView

Figure 40. Toolbar Package Classes

A.7 workspace Package

MyGlassPane (JComponent)

-attribute1:int
#keyListener:GPKeyListener=null
#m_bean:Object=null
#m_id:Integer
#m_point:Point
#mouseListener:GPListener=null

+MyGlassPane(view:Component,b
+stateChanged(bean:Object):void
+updateBean(bean:Object,view:Co

GPKeyListener
GPListener

BeanListener (UnicastRemoteObject / RemoteEventListener)

-attribute1:int
-m_bean:BeanData
-m_name:String
-m_space:JavaSpace
-m_workspace:Workspace

+BeanListener(bean:BeanData,
+BeanListener()
+notify(ev:RemoteEvent):void

space:JavaSpace

WorkspaceContents (Serializable)

-m_beans:Map
-m_name:String

+WorkspaceContents(name:Str
+WorkspaceContents()
+addBean(Id:Integer,beanData:E
+addBean(beanData:BeanData)
+getBean(Id:Integer):BeanData
+removeAllBeans():void
+removeBean(Id:Integer):void

beans:Map
name:String

BeanData (JInternalFrame / Serializable)

-DEBUG:boolean=true
-attribute1:int
-m_awareGlassPane:AwareGla
-m_bPointer:JButton
-m_bean:Object
-m_beanIcon:ImageIcon
-m_beanListener:BeanListener
-m_button:JButton
-m_desktop:Desktop
-m_glassPane:MyGlassPane=n
-m_ib:JButton=Standard.Button(
-m_id:Integer
-m_index:int
-m_jarInfo:IJarInfo
-m_radarview:RadarView
-m_scrollPane:JScrollPane
-m_userId:String
-m_view:Component
-m_workspace:Workspace

+BeanData(bean:Object,ji:IJarIn
+BeanData()
+repaint():void

-ButtonHandler
-InternalFrameHandler
-PointerHandler
-Support
-UserColorHandler

ID:Integer
awareGlassPane:AwareGlassF
bean:Object
beanImageIcon:ImageIcon
beanName:String
buttons:JButton[]
index:int
jarInfo:IJarInfo
maximumSize:Dimension
radarView:RadarView
view:Component
viewportBorderBounds:Rectang
workspace:Workspace

Workspace (JFrame)

-DEBUG:boolean=true
+MAX_SIZE:Dimension=new Di
-m_beanDatas:Map=new Hash
-m_demuxer:Demuxer=new Der
-m_desktop:Desktop
-m_desktopPane:DesktopPane
-m_name:String
-m_session:ISession=null
-m_tool:Tool=null
-m_toolBar:JToolBar=Standard.
-m_workspaceContentsListener

+Workspace(session:Session,r
+addBean(ji:IJarInfo,index:int):vo
+addBean(ji:IJarInfo,beanName
+addBeanData(bd:BeanData):vo
+connect():void
+disconnect():void
+distributeBean(sessionName:
+getBeanData(bdId:Integer):Bea
+getWorkspaceContentsBeanD
+rebuildWorkspace():void
+redisplay():void
+removeBean(bd:BeanData):voi
+removeBeanData(bd:BeanData

#BeanRemoveHandler
-BeanToolbarHandler
-DesktopPane
#MouseHandler
-TestHandler
-Tool
-UserColorHandler
-WindowHandler

beanDataMap:Map
beanDatas:Collection
demuxer:Demuxer
desktop:Desktop
desktopPane:JDesktopPane
maximumSize:Dimension
name:String
session:ISession
toolBar:JToolBar
workspaceContentsBeanData:C
workspaceContentsBeanDataT

WorkspaceContentsListener (UnicastRemoteObject / RemoteEventListener)

+notify(ev:RemoteEvent):void

space:JavaSpace

0..*

WorkspaceContents contains only BeanDataIDs not the actual BeanData object.
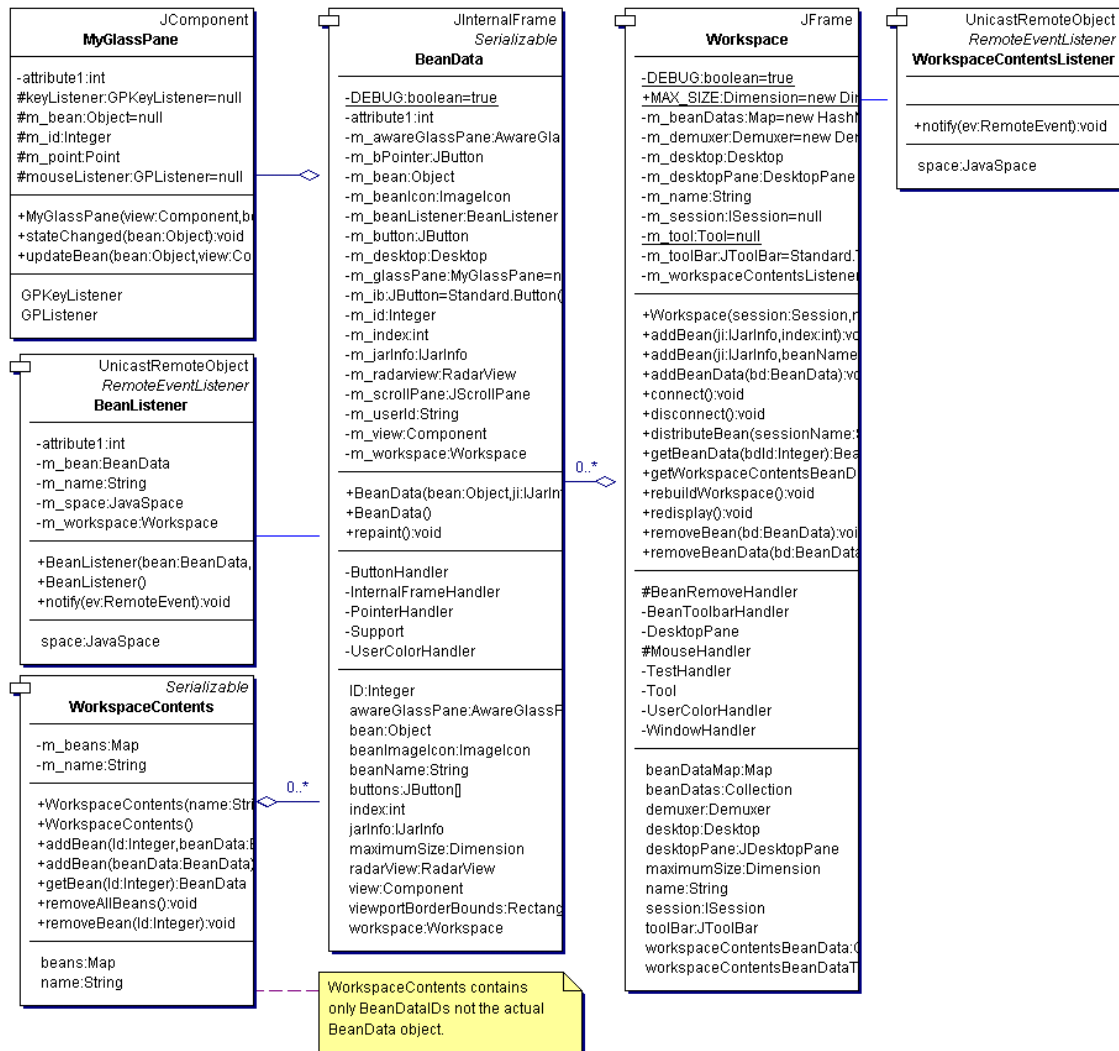
Figure 41. Workspace Classes

104

B. Appendix B - RMI Information

This appendix covers the basics of Remote Method Invocation (RMI). This is not intended to provide a deep knowledge and cover the topic in depth, but should help provide additional background information on what RMI is. Jini™ technology relies heavily on RMI system for its implementation. Additional information can be found in RMI [54] and Appendix A of Core Jini [16].

B.1 Overview

Remote Method Invocation (RMI) provides the mechanisms needed for applications running on different Java Virtual Machines (JVMs) to communicate with each other. Very much like Remote Procedure Call (RPC), a well-known concept in distributed systems, RMI enables one application to call another class' method on another machine.

B.2 Remote Interfaces

Remote objects are objects that expose their methods so they can be called by other objects on other machines. In client/server terminology, the remote object is the server and the caller of the method is the client. Note that RMI is more robust than strict client/server architectures. *Remote* objects can have element of both client and server within the same object.

For a client to make a call to the *remote* object, it first must know what interface it implements. Thus, providing the client information on what methods are available on the server. RMI provides a set of Application Programmer Interfaces (APIs) called the server's *remote interface*. This interface defines what methods can be invoked from outside the server's JVM.

The remote interface of a server is simply defined by any interface that extends the RMI *Remote* interface. This acts as a flag to RMI to indicate that this interface must have the mechanism implemented to enable its methods to be called remotely. The server object that actually does the work is just the implementation of this remote interface.

B.3 Stubs and Skeletons

The JVM only knows how to perform local method invocations. Thus, to enable remote method invocations, a bit of extra code on top of the JVM must be used.

The first functionality needed to be added is for the server to be able to handle network connections from clients, read data from those clients, and then turn that data into a local method invocation. Second is for the client to make a local call on an object that represents the remote object. (They must both directly invoke methods on local objects, because that is all the JVM knows how to do.) This local object then creates the connection with the server and passes it data. Once the invocation completes, the server must send the data back to the client.

On the server side, two classes handle most of the additional functionality. First, to provide network communications, the server object extends a class called RMI *UnicastRemoteObject*. This extension has all the low level detail necessary to send and receive messages. The second class, called the *skeleton*, handles the calling of a specific method on a particular server object. The *skeleton* object is paired with a server object, because it needs to know what methods are available, and what parameters and return value they have. The *skeleton's* main job is to take data received from the network, figure out what operation to invoke on the server and return the result.

The client side is a little simpler, because all the client has to do is map the local object invocation on the object representing the server to actual network communication. The local object that handles the responsibility of packaging data and managing the network connection is called the *stub*. So, whenever a client makes a remote invocation, it actually invokes a method in the local *stub* object. The *stub* then sends a message to the remote JVM, where it is received and translated by the *skeleton* into a local method call. Figure 42 illustrates the how the client and the server communicate through the use of *stubs* and *skeletons*. This may seem like a lot of work, but most of it is transparent to the users. So when programming the client, just write the code in terms of the remote interfaces that will be call.
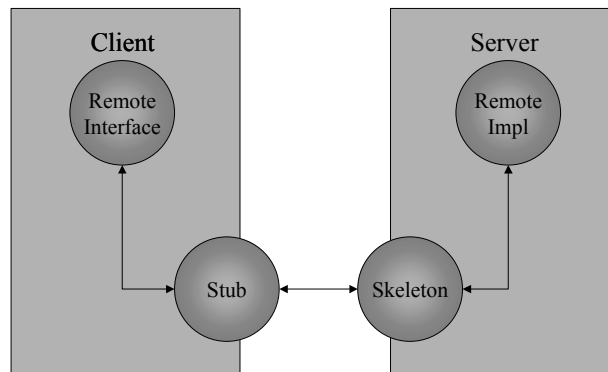
Figure 42. RMI - Client and Server Communicate via Stubs and Skeletons [16]

B.4 Serialization

Stub and skeletons take care of packaging the data to be sent over the network. The network only understands one thing, and that is bytes. So, the stubs and skeletons take care of turning the parameter and return value into streams of bytes and then reconstituting them on the other end. The mechanism for doing this transformation into a stream of bytes is called *serialization*.

The Java language defines an interface called *Serializable*. This interface does not add any additional methods that are required to be implemented, but acts as a tag to indicate to Java that this class may be serialized.

Without this mechanism of serialization, RMI would have no way of sending complex objects over the network. Any object that is used as a parameter or return value must be serializable. Primitive types (e.g. int, boolean, long) are considered to be serializable.

For an object to become serializable several condition must exist. First, it must have a public no-argument constructor defined. This is needed in the deserialization process. Second, the class may not reference any non-serializable objects. If it does reference an object that is not serializable, this object becomes non-serializable.

B.5 Parameters and Return Values

How does parameter passing in RMI actually work? Any parameter or return value that is used in an RMI must be serializable. The fact that all the data is serialized prior to being sent has some important ramifications. In local method calls, references to objects are passed. However, RMI copies the arguments and return values of the remote calls. So the semantics for input and output objects are "pass by value" rather than "pass by reference."

What happens when an application passes a remote object into a remote method as an argument or returns it from a remote method? To answer this question, RMI does some "under the covers" operation to handle this situation. Say in the implementation of the remote server, one of its methods returns a reference to *this* (or in other words, itself). On the server, *this* refers to the actual implementation object that lives in the server's virtual machine (VM). Remember that the client always deals with the server through its stub; thus, it has no way of directly referring to an object on another VM. So what is desired when the server returns itself, is for the client to receive a reference to the server's stub. Thus, it gives the appearance that the client is working directly with the remote object.

To make this happen, RMI searches for input and output parameters that are references to objects that implement the *Remote* interface. When one of these is identified, it replaces the stub as appropriate. So if a server returns a reference to itself, RMI "converts" it to a stub so the client can use it.

This transparent argument swapping gives the illusion that both the server and client are working with local objects. This, in effect, maintains the "pass by reference" semantics. When the application passes a remote object in or out of a method call, what actually is returned is a "live" reference to that remote object.

B.6 Dynamic Code Loading

Serialization only packages up member data within an object; it does not package up the code that implements the object. So if an object is sent over a network, how can it be used if the other end does not

get the code? The answer to this question is what sets RMI apart from traditional remote procedure call. RMI allows a JVM to dynamically download implementation files when needed.

An example will best illustrate this concept. Consider a server that implements a sort routine that takes as a parameter a *List* (well-defined interface that is part of Java 2) and returns a *List*. Since this is an interface, the client can send a class that implements that *List* interface. Now consider the client wants to use a new, custom implementation called *QuickList*. This works fine on a local system because the sort routine only cares that the input speaks the *List* interface and the implementation of the *QuickList* is available. This is a good example of polymorphism at work.

However, this situation presents a problem in the remote case. The server knows nothing about this *QuickList* class and does not have the implementation available to it. To solve the problem, it would be efficient to send the *QuickList* implementation to the server so it can operate on the new data structure and return a result. RMI does exactly that: it sends the implementation of unknown classes to enable servers to operate on them.

Normally, a Java application finds all the needed implementations of classes in it *classpath* – a set of directories or JAR files containing class files. RMI extends the concept of the *classpath* with the notion of a *codebase*. The *codebase* can be thought of as a new location for classfiles that is dynamically allocated so a program can retrieve implementation to new classes.

Any program that exports classes sets a codebase that indicates where the implementations may be found. The codebase is then sent to any downloading program tagged on the serialization of the object's data. The receiver then reconstitutes the serialized object and, if the classfile is not locally available, downloads it from the location indicated by the codebase. RMI uses the ability of Java programs to copy bytecodes from URLs and securely execute them to provide downloadable code.

A hypertext transfer protocol (http) server most commonly services the capability of downloadable code. Programs that export code set a codebase, via a property to the server's JVM, that contains the http URL that indicates the location of the codebase. The most common use of downloadable code is the

transmission of remote object stubs to clients. This way the client only needs to know about the remote interface that the server object implements.

More information regarding codebase can be found in a chapter called "How Codebase Works" in Core Jini, Second Edition written by Keith W. Edwards [15].

B.7 Security Concerns

Being able to download code from another application on the fly creates some serious security concerns. What if the code that is downloaded is malicious? Just like applets, RMI can provide a restricted environment for running code obtained in this fashion.

In Java, a *SecurityManager* installed in the run time environment maintains application security. To prevent malicious code from doing harm, RMI will not run any code if there is no *SecurityManager* active in the downloading program – instead, the program must be able to find all classes, including stubs in the local classpath.

This situation, however, is not suitable for production. RMI provides a simple security manager that can be set to run, to enable downloadable code. It is called *RMISecurityManager*. A security policy file that is passed in on the command line to a Java program configures security for the VM. The security policy file defines certain permission for the code in an application, based on where the code came from.

B.8 Building, Compiling and Running RMI Programs

When building an RMI program, the first thing to do is write a remote interface that extends the *Remote* interface. Next supply an implementation to the remote interface and a client that will use the remote interface to make the method invocation. For illustration, three java files named: Add.java, AddImpl.java, and AddClient.java are created. The Add.java file provides the remote interface. The AddImpl.java file contains the implementation or the server code of an array adder. The last file, AddClient.java, is the client code that will use the remote interface to add the contents of two arrays. Once all the code is written, they need to be compiled using the Java compiler.

javac Add.java AddImpl.java AddClient.java

This command above will compile all the java files and output the appropriate bytecode class files. Currently, no mention of stub or skeletons files has been made. So where do they get created? RMI provides a stub compiler that is used to generate the stubs and skeletons called *rmic*. Thus, the following command would need to be issued on the class that implements the remote interface.

rmic AddImpl

This command will generate two files, namely *AddImpl_Stub.class* and *AddImpl_Skel.class*. These are then used to enable the RMI to occur.

To run a RMI application, the RMI registry process must be started first. This is started by the *rmiregistry* command. The server application is then run and bound it to the RMI registry. A simple command such as: *java AddImpl* does the trick. This must be done so the client will have a server to access when it makes its remote procedure call. With all that accomplished, the application can finally be run with the simple command: *java AddClient*. The *AddClient* class will then use the *AddImpl* server to add two arrays and send back a result.

Bibliography

1. Alexander, Christopher, and others. <u>A Pattern Language</u>. New York: Oxford University Press, 1977.

2. Boyd, John R. <u>A Discourse on Winning and Losing</u>. Report No: MU43947. Set of briefing slides available at Air University Library, Maxwell AFB, AL, August 1987.

3. Butler, Sean C. <u>A Flexible Framework for Collaborative Visualization Applications Using JavaSpaces™</u>. MS thesis, AFIT/GCE/ENG/01M-01. Graduate School of Engineering and Management, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, March 2001 (ADA391953).

4. Card, S. K. and others. <u>Readings in Information Visualization: Using Vision to Think</u>. San Francisco: Morgan Kaufmann Publishers, Inc., 1999.

5. Ceglar, Aaron, and Paul Calder. "A New Approach to Collaborative Frameworks using Shared Objects," <u>Proceedings of the Computer Science Conference, ACSC 2001</u>. 3-10. 24th Australasian, 2001.

6. Chabert, Annie, Ed Grossman, Larry Jackson, and Stephen Pietrovicz, "NCSA Habanero – Synchronous Collaborative Framework and Environment." White Paper, NCSA, Champaign IL, 1998.

7. Chabert, Annie, Ed Grossman, Larry S. Jackson, Stephen R. Pietrowiz, and Chris Seguin. "Java Object-Sharing in Habanero," <u>Communication of the ACM, 41, 6</u>: 69-76 (June 1998).

8. Chuah, Mei C. and Stephen G. Eick. "Information Right Glyphs for Software Management Data," <u>IEEE Computer Graphics and Applications</u>: 24-29 (July/August 1998).

9. Cowan, Jeffery L. <u>From Air Force Fighter Pilot to Marine Corps Warfighting: Colonel John Boyd, His Theories on War, and their Unexpected Legacy</u>. Thesis. United States Marine Corp Command and Staff College, Quantico VA, 2000.

10. Department of Defense. <u>America's Air Force Vision 2020</u>. AF Vision 2020. Washington DC: HQ USAF. http://www.af.mil/vision.

11. Department of Defense. <u>Joint Vision 2020</u>. Chairman Joint Chief of Staff. http://www.dtic.mil/jcs/. 20 September 2001.

12. Department of the Army, and U.S. Marine Corps. Headquarters. <u>Operational Terms and Graphics</u>. FM 101-5-1/MCRP 5-2A Washington: HQ USA and HQ USMC, 30 September 1997.

13. Deutsch, Peter. "The Seven Fallacies of Distributed Computing." Excerpts from internal Sun Microsystems meetings. http://java.sun.com/people/jag/Fallacies.html. 9 January 2002.

14. Driggers, Brent, Jay Alameda, and Ken Bishop. "Distributed Collaboration for Engineering Scientific Applications," <u>ACM 1997 Workshop on Java for Science and Engineering Computations</u>. Las Vegas, NV USA, 21 June 1997.

15. Edwards, W. Keith. "Jini Planet: How Codebase Works." Excepts from Core Jini Second Edition. http://www.kedwards.com/jini/codebase.html. 5 December 2001.

16. Edwards, W. Keith. <u>Core Jini™</u>. New Jersey: Prentice Hall, 1999.

17. Ellis, C., and S. Gibbs. "Concurrency Control in Groupware Systems," <u>Proceeding of ACM SIGMOD</u>. 399-407. 1989.

18. Fayad, M. and others. <u>Building Application Frameworks</u>. New York: Wiley, 1999.

19. Freeman, Eric and others. <u>JavaSpaces Principles, Patterns and Practice</u>. Massachusetts: Addison-Wesley, 1999.

20. Gamma, Erich and others. <u>Design Patters Elements of Reusable Object-Oriented Software.</u> Massachusetts: Addison-Wesley, 1995.

21. Gutwin, Carl and Saul Greenberg. "Design for Individuals, Design for Groups: Tradeoffs between Power and Workspace Awareness," <u>Proceedings of ACM CSCW '98</u>. Seattle WA, 1998.

22. Gutwin, Carl and Saul Greenburg. "The Mechanics of Collaboration: Developing Low Cost Usability Evaluation Methods for Shared Workspaces," <u>Proceedings of IEEE 9th International Workshops on WEB Enabling Technologies: Infrastructure for Collaborative Enterprises, 2000 (WET ICE 2000)</u>. 98-103. Gaithersburg MD, 14 Jun - 16 Jun 2000.

23. Gutwin, Carl, and Saul Greenberg. "Workspace Awareness for Groupware," <u>Proceeding of Computer-Human Interaction Conference (CHI 96)</u>. 1996.

24. Gutwin, Carl, Saul Greenberg, and Mark Roseman. "Workspace Awareness Support with Radar Views," <u>Proceeding of Computer-Human Interaction Conference (CHI 96)</u>. 1996.

25. Hammond, Grant L. "The Essential Boyd." Unpublished Paper. Director of the Center for Strategy and Technology, Air War College, Maxwell AFB AL, 2001. http://www.belisarius.com/modern_business_strategy/hammond/essential_boyd.htm.

26. Holzhauer, Douglas and others. "Building an Experimental Joint Battlespace Infosphere (YJBI-CB)." Report to AFRL/IFTC. Air Force Research Laboratory/ Information Directorate, Rome NY, 2001.

27. Jacobs, Timothy M and Sean C. Butler. "Collaborative Visualization for Military Planning," <u>Proceedings of the SPIE ITCOM 2001</u>. Denver CO, 2001.

28. Juth, Stephen. <u>Collaboration Components for Programming Real-time Synchronous Groupware Applications</u>. MS Thesis. Graduate School-New Brunswick, Rutgers New Jersey, October 1998.

29. Khetawat, Amit Kumar. <u>Thesis on Collaborative Computing On the Internet</u>. Masters Thesis. Department of Electrical and Computer Engineering, North Carolina State University, Raleigh NC, May 1997.

30. Koch, M., D. Kohler, and M. Burger. "Awareness Information in Wide Area Network." Technical Report. Applied Informatics and Distributed Systems Group, Depart of Informatics, Technische Universitat Munchen, 1996.

31. Krasner, G. E. and S. T. Pope. "A Cookbook for using the Model-View-Controller User Interface Paradigm in Smalltalk," <u>Journal on Object Oriented Programming</u>, (August/September 1988).

32. Li, Wen, Weicong Want, and Ivan Marsic. "Collaboration Transparency in the DISCIPLE Framework," <u>Proceeding of the International ACM SIGGROUP Conference on Supporting Group Work</u>. 326-333. 1999.

33. Marsic, Ivan. "DISCIPLE: A Framework for Multimodal Collaboration in Heterogeneous Environments." <u>ACM Computing Surveys, Vol 31 Issue 2es, Article 4</u>, (1999).

34. Maurer, Martha E. "Coalition Command and Control." National Defense University, Fort McNair, Washington DC, 1996.

35. McCarthy, J. and others. <u>Building the Joint Battlespace Infosphere, Volume 1: Summary</u>. Research Report SAB-TR-99-02, USAF Scientific Advisory Board, December 17 1999.

36. McQuay, William K. "Distributed Collaborative Environments for the 21st Century Engineer." <u>Proceedings of the IEEE 2000 National Aerospace and Electronics Conference, 2000. NAECON 2000</u>. 407-414. 10-12 October 2000.

37. MindSim Corp. http://www.mindsim.com/MindSim/Corporate/OODA.html. September 2001.

38. Mucks, J. H., and L. A. Jesse. "Web-Enabled Timeline Analysis System." White Paper. http://webtas.com/white-papers.shtml. 17 January 2002.

39. Munson, J. and P. Dewan. "A Concurrency Control Framework for Collaborative Systems." Technical Report, Department of Computer Science, University of North Carolina, 1996.

40. "NCSA Habanero", NCSA website. http://havefun.ncsa.uiuc.edu/habanero. 20 September 2001.

41. Özsu, M. Tamer, and Patrick Valduriez. <u>Principles of Distributed Database Systems 2nd. ed</u>. Prentice Hall, Inc., ISBN 0-13-659707-6, 1999

42. Paranj, Bala. "Java Tip 68: Learn how to implement the Command pattern in Java", Java World Tip website. http://www.javaworld.com/javaworld/javatips. 5 February 2002.

43. Patterson, John F., and others. "Rendezvous: An Architecture for Synchronous Multi-User Applications," <u>Proceeding of the CSCW 90</u>. 317-328. October 1990.

44. "Requirements for an Air Force Collaborative Enterprise Environment, Version 2." Paper on Collaborative Technology Development. Air Force Research Laboratory, Wright-Patterson AFB OH, January 2000.

45. Roberts, Simon and Jon Byous. "Distributed Events in Jini™ Technology." Paper published on Java Developer Connection™. http://java.sun.com. June 1999.

46. Roussev, Vassil and others. "Composable Collaboration Infrastructure Based on Programming Patterns," Proceeding of the ACM Conference on Computer Supported Cooperative Work (CSCW), 2000. 117-126. Philadelphia PA. 2-6 December 2000.

47. Schuckmann, Christian and others. "Designing Object-Oriented Synchronous Groupware with COAST," Proceedings of the ACM 1996 Conference on Computer Supported Cooperative Work. 30-38. Boston MA, 16-20 Nov 1996.

48. Scientific Advisory Board, United States Air Force, "Building the Joint Battlespace Infosphere." SAB-TR-99-02. December 17, 1999.

49. Stang, Mark and Stephen Whinston. "Enterprise Computer with Jini™ Technology." IT Professional, Vol. 3 No. 1: 33-38 (January/February 2001).

50. Sun Microsystems, Inc. "How to User Root Panes." Excerpt from The Java™ Tutorial; Trail: Creating a GUI with JFC/Swing; Lesson: Using Swing Components. http://java.sun.com/docs/books/tutorial/uiswing/components/rootpane.html. 9 January 2002.

51. Sun Microsystems, Inc. "JavaBeans™ - The Only Component Architecture for Java™ Technology." Except from JavaBeans™ web resources. http://java.sun.com/products/javabeans. 3 January 2002.

52. Sun Microsystems, Inc. JavaBeans™ 1.01 API Specification. Mountain View CA, July 1997.

53. Sun Microsystems, Inc. JavaSpaces™ Service Specification Version 1.1. Palo Alto CA, October 2000.

54. Sun Microsystems, Inc. Java™ Remote Method Invocation Specification. Revision 1.50. JDK 1.2. Mountain View CA, October 1998.

55. Sun Microsystems, Inc. Jini™ Technology 1.1 API Documentation. Mountain View CA, 1999.

56. Sun Microsystems, Inc. Jini™ Technology Core Platform Specification. Version 1.1. Mountain View CA, 2000.

57. United States Air Force Scientific Advisory Board. Information Management to Support the Warrior. Report SAB-TR-98-02, December 1998.

58. Wang, Weicong, and others. "Design of the DISCIPLE Synchronous Collaboration Framework," Proceeding of the 3rd IASTED International Conference Internet and Multimedia Systems and Applications. 316-324. Nassau, Grand Bahamas. 18-21 October 1999.

59. Wood, Jason. <u>Collaborative Visualization</u>. Ph.D. dissertation. School of Computer Studies, University of Leeds, Leeds UK, 1998.

Vita

Captain Chad M. Harris spent many years an Army dependent. His family settled down in Bountiful, Utah where he graduated from Viewmont High School in May 1991. He then pursued his studies in Computer Science at Brigham Young University (BYU) in Provo, Utah. He served a mission for the Church of Jesus Christ of Latter-day Saints from May 1992 to May 1994 in Osorno, Chile. Upon return from his mission he met his wife and was married 15 June 1995. Later he completed his undergraduate education and graduated with a Bachelor of Science degree in Computer Science and was commissioned in December 1997.

In route to his first assignment he went to Keesler AFB, Biloxi Mississippi for Basic Communication Officer Training (BCOT). Following graduation from BCOT he was assigned to Falcon AFB, which later changed its name to Schriever AFB, as Chief of the Software Configuration Management shop. There he was entrusted to be the Mission Support Flight Commander and served in that capacity until his selection to the Graduate School of Engineering and Management, Air Force Institute of Technology (AFIT). Upon graduation from AFIT, he will be assigned to the 690th Computer System Squadron at Lackland AFB, San Antonio, Texas.

1. REPORT DATE *(DD-MM-YYYY)* 05-03-2002	2. REPORT TYPE Master's Thesis	3. DATES COVERED *(From - To)* Mar 2001 - Mar 2002

4. TITLE AND SUBTITLE A COLLABORATIVE VISUALIZATION FRAMEWORK USING JINI™ TECHNOLOGY	5a. CONTRACT NUMBER
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) Harris, Chad M., Captain, USAF	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Air Force Institute of Technology Graduate School of Engineering and Management (AFIT/EN) 2950 P. Street, Building 640 WPAFB OH 45433-7765	8. PERFORMING ORGANIZATION REPORT NUMBER AFIT/GCS/ENG/02M-04

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Air Force Research Laboratory/IFTC (AFMC) Attn: Douglas Holzhauer, Ph.D. 26 Electronics Parkway Rome, NY 13441-4514 Comm: (315) 330-4920 DSN: 587-4920 Email: douglas.holzhauer@rl.af.mil	10. SPONSOR/MONITOR'S ACRONYM(S) AFRL/IFTC
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

13. SUPPLEMENTARY NOTES

Lt. Col. Timothy M. Jacobs, ENG, (937) 255-6565 x4279, Timothy.Jacobs@afit.edu

14. ABSTRACT
    It is difficult to achieve mutual understanding of complex information between individuals that are separated geographically. Two well-known techniques commonly used to deal with this difficultly are collaboration and information visualization. This thesis develops a generic flexible framework that supports both collaboration and information visualization. It introduces the Collaborative Visualization Environment (COVE) framework, which simplifies the development of real-time synchronous multi-user applications by decoupling the elements of collaboration from the application. This allows developers to focus on building applications and leave the difficulties of collaboration (i.e. concurrency controls, user awareness, session management, etc.) to the framework.
    The framework uses an object sharing approach to share information and views between participants in a collaborative session. This approach takes advantage of several Java technologies (i.e. JavaBeans™, Jini™, and JavaSpaces™). JavaBeans™ establish a well-known standard for applications to operate within the framework. Jini™ services provide framework stability and enable code sharing across the network. Objects are shared between remote clients through the JavaSpaces™ service.

15. SUBJECT TERMS
Collaboration, Information Visualization, Framework, Software Architectures, Jini, JavaSpaces, JavaBeans

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON Lt. Col. Timothy M. Jacobs
a. REPORT	b. ABSTRACT	c. THIS PAGE			
Unclassified	Unclassified	Unclassified	UU	130	19b. TELEPHONE NUMBER *(Include area code)* (937) 255-6565 x4279